UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

PESTT:
PESTT EDUCATIONAL SOFTWARE TESTING TOOL

Rui Manuel da Silveira Gameiro

PROJECTO

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Engenharia de Software

2012

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

PESTT:
PESTT EDUCATIONAL SOFTWARE TESTING TOOL

Rui Manuel da Silveira Gameiro

PROJECTO

Projecto orientado pelo Prof. Doutor Francisco Cipriano da Cunha Martins

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Engenharia de Software

2012

Agradecimentos

Agradeço em primeiro lugar ao meu orientador o Professor Doutor Francisco Martins, pelo apoio, suporte e dedicação demonstrada durante este último ano. Foi um privilégio e um grande prazer poder realizar este projecto juntamente com o Professor. Mais do que meu orientador, considero o Professor um verdadeiro amigo. Muito Obrigado Professor.

Um especial agradecimento também ao meu colega João Veiga, pelas informações fornecidas que foram fundamentais para que este projecto avançasse, pelas conversas que tivemos durante os almoços, essenciais não só para nos distrairmos, mas também para trocarmos ideias.

E o que seria de mim sem a minha família? Queria portanto agradecer a quem mais fez por mim, aos meus pais e a minha irmã. Aos meus pais por me darem todas as condições e suporte necessários para eu ter conseguido chegar onde cheguei, pelos momentos que dediquei mais ao trabalho e menos a vocês, por me aturarem, pela vossa dedicação, pelo vosso apoio incondicional e, principalmente, pela vossa paciência, o meu Muito Obrigado por tudo o que fizeram por mim. Obrigado também à minha irmã, pelo apoio e pela ajuda que me deu nos momentos mais difíceis.

Um obrigado ao resto da minha família e amigos pelos bons momentos que passámos juntos.

Espero que o meu esforço, dedicação, empenho e resultados tenham superado as vossas expetativas.

A todos aqueles que permitiram que tudo isto fosse possível,

Muito Obrigado.

Resumo

A sociedade em que vivemos é completamente dependente da tecnologia. Desde os simples equipamentos domésticos, como televisores, frigoríficos ou telefones, passando, por exemplo, pelos meios de transporte (e.g., automóveis, comboios, aviões), todos eles contêm *software* que é indispensável ao seu funcionamento, e que se falhar pode ter consequências devastadoras, tanto em termos da nossa comodidade como em impacte financeiro.

A ubiquidade e consequente criticidade do *software* requer que este possua um nível de qualidade elevado. É neste contexto que entram os *testes de software*, que constituem um dos principais mecanismos para reduzir a ocorrência de problemas e garantir a qualidade do *software* produzido.

Saber planear, desenhar e automatizar a execução de testes é portanto uma valência imprescindível ao engenheiro de *software*. A ferramenta que propomos—*PESTT Educational Software Testing Tool (PESTT)*—vem auxiliar a atividades de desenho e de análise de cobertura de testes unitários baseados em grafos de controlo de fluxo (CFG).

A ferramenta inicialmente foi pensada para ser usada como suporte ao ensino dos conceitos e técnicas introdutórias de teste de *software*, mas pode também ser usada em outros cenários. O UI cuidadosamente pensado e a sua integração flexível no *Eclipse* IDE, torna o *PESTT* um auxiliar valioso para professores e alunos.

Os conceitos de testes encontram-se bem organizados: uma vista para o CFG (automaticamente gerado a partir do código fonte *Java*), uma vista onde pode ser escolhido o critério de cobertura estruturada de uma maneira que torna simples visualizar que critério *subsume* outros, uma vista para gerir os requisitos de teste e a sua relação com o CFG e com o código fonte, uma vista para os caminhos de teste, uma outra para as estatísticas de cobertura e ainda uma vista para a informação relacionada com o fluxo de dados. Todas estas vistas encontram-se integradas umas com as outras de forma a poder disponibilizar a maior quantidade possível de informação ao engenheiro de teste.

O *PESTT* encontra-se integrado com o *JUnit* e com o *Byteman* de forma a poder disponibilizar os caminhos executados (no CFG) pelos testes; disponibiliza também as estatísticas referentes a cada teste ou para o conjunto de testes, que permite que o engenheiro possa ter a perceção dos níveis de cobertura atuais (individual ou total) dos testes.

O *PESTT* distingue-se das outras ferramentas de teste de análise de cobertura, porque não oferece apenas análise de cobertura baseada em nós ou arestas. Oferece entre outras funcionalidades análise de cobertura baseada em CFG (que para além de incluir os critérios de cobertura de nós e arestas) inclui outros mais poderosos e auxilia o engenheiro de testes no planeamento das várias atividades do processo de testes.

Este documento dá a conhecer os aspetos relacionados com o *PESTT*, necessários para a sua concretização, noemadamente, os conceitos teóricos essenciais para a sua correta utilização, as ferramentas de cobertura existentes e as respetivas contribuições para o *PESTT*, passando pela arquitetura, não esqueçendo a implementação dos algoritmos dos critérios de cobertura baseados no CFG, fundamentais para obter os requisitos de teste, na obtenção dos caminhos de testes executados, entre outras funcionalidades.

Palavras-chave: *PESTT, Plug-in, Eclipse, Teste de Software, Ensino.*

Abstract

The society we live in is completely dependent of technology. From simple home appliances such as televisions, Refrigerators or telephones, passing, for example, by transports (e.g., cars, trains, plains), they all contain software that is essential to its operation, and if it fails the consequences can be devastating, both in terms of our comfort as financial impact.

The ubiquity and consequent criticism of Software requires that this have a high level of quality. It is in this context that enter the Software tests, which are a major mechanism for reducing the occurrence of problems and ensure the quality of the Software produced.

Learn to plan, design and automate test execution is therefore a valence essential to the Software engineer. The tool we propose—*PESTT Educational Software Testing Tool (PESTT)*—comes to auxiliary the activities of design and coverage analysis of unit testing based on flow control graph (CFG).

PESTT started as a tool specially tailored for teaching how to test software, but can be very well used in other scenarios. Its suits well the teaching of software testing classes, because of its careful designed UI and its smoothly integration with the *Eclipse* IDE.

The testing concepts are well organized: a view for the control flow graph (automatically generated from the method's source code), a view with the supported coverage criteria arranged in such a way that it is easy to visualize which criteria subsumes others, a view for manipulating test requirements and its explanation both in terms of the control flow graph and of the source code, a view for the test paths, another for coverage statistics, yet another for data flow information, and all these views integrated together to give as much information as possible to the test engineer.

It provides integration with *JUnit* and *Byteman* and reconstructs run paths of each test method, computing statistics either for each test method or for the complete test set, this allows the engineer to get the current levels of tests coverage (individual or total).

PESTT distinguishes from other coverage analysis testing tools because it is not just a node and branch coverage analyzer. Among other things, it supports many other more powerful coverage criteria and assists the test engineer in the various tasks of the testing activity.

This report aims at describing the aspects of *PESTT*, citing the necessary mechanisms for its implementation, in particular, theoretical concepts used essential for its correct use,

the existing coverage tools and their contributions to *PESTT*, through the architecture, not forgetting the implementation of the algorithms of coverage criteria based CFG fundamental to obtain the test requirements or obtaining the executed paths of tests performed, among other features.

Keywords: *PESTT, Plug-in, Eclipse, Software Testing, Education*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software Engineering is an area of Computer Science that uses a set of management processes, design activities, and software tools, whose goal is the development of software. The main focus of Software Engineering is to create good solutions for computing problems and for information processing:

- with quality;
- meeting deadlines;
- meeting the needs of customers and users;

through the application of systematic and disciplined approaches of development and maintenance. In other words, Software Engineering can be defined as a discipline that applies Engineering principles in order to produce quality software (the product).

In a society completely dependent on technology, from simple home appliances, such as televisions, telephones, and refrigerators, to, for example, transport system (e.g., cars, trains, planes), they all contain software, which is essential to its operation, and if it fails, the consequences can be devastating, both in terms of our comfort and of financial impact. The ubiquity and consequent criticality of software requires that it must have a high quality level. To ensure the final quality of the developed software, it is necessary to use different processes, techniques, methodologies, and tools in the various stages of the software development process. Is in this context that Software Testing appear, as one of the activities of the development process, whose purpose is to minimize the existence of software failures and to ensure the quality of the produced software.

Learn how to plan, design, and automate the execution of tests is therefore essential to a software engineer, which needs to master the different techniques and concepts associated with software testing.

Currently, there are several tools available on the market in the *Verification, Validation, and Testing* area, providing many features; usually they are paid and very complicated to handle. Free tools tend to provide less features than the commercial ones, generally offer only one specific type of test/validation, which make them very limited.

However, none of these tools was designed to be used in teaching Software Testing, where the techniques and basic concepts are teached. Is in this context that *PESTT Educational Software Testing Tool (PESTT)* appears. This thesis describes *PESTT*, its architecture and its internals, and try to highlight the benefits of using *PESTT* in teaching the techniques and concepts of Verification of Software Testing.

The present work was motivated by the following remarks:

- the increasingly importance of Software Testing in the present days;
- the benefits that the automatization of the test design and test analysis processes have in terms of time required and occurrence of errors;
- the impact in learning Software Testing techniques and concepts when using a tool specifically designed for teaching;
- the influence of the tool in the quality improvement of the produced software;
- my interest in the Software Engineering area, especially in Software Testing.

The objective of this project is the production of a software tool to support teaching, in particular, the introduction of basic concepts and of the different techniques of Software Testing. In order to develop a useful and easy to use tool, some requirements need to be met:

- determine the requirements for unit testing, based on methods written in the Java programming language;
- apply a coverage criterion selected by the user;
- make available an integrated interface with the *Eclipse* [14] IDE *(Integrated Development Environment)*, to help plan and view the tests run or to performed.

The main contributions of the work described in this thesis comprise:

(i) the implementation of the project basis through:

  (a) a flexible integration in the *Eclipse* IDE;

  (b) a flexible integration with *Zest* visualization tool;

  (c) a UI implementation.

(ii) the mapping of a method source code with a Control Flow Graph (CFG), and vice-versa;

(iii) the implementation of different types of operations on the CFG:

  (a) its optimization;

  (b) adding additional information to it;

  (c) linking it with the source code.

(iv) the implementation of coverage criteria algorithms based on CFGs;

(v) the generation of test requirements for a method;

(vi) a flexible integration with *JUnit*;

(vii) a flexible integration with *Byteman*;

(viii) the presentation of visual coverage status information.

This work took place at the Large-Scale Informatics Systems Laboratory (LaSIGE-FCUL), a research unit of the Department of Informatics (DI) of the University of Lisbon, Faculty of Sciences. It was developed within the scope of the *PESTT* project, which follows the Software Testing guidelines by *Paul Ammann and Jeff Offutt* [30].

The remainder of this thesis is structured as follows. **Chapter** 2 reviews the basic underlying concepts behind *PESTT*. **Chapter** 3 presents some tools used during the course of this work and points related tools for Software Testing. **Chapter** 4 describes *PESTT* architecture and **Chapter** 5 the development details of *PESTT*. **Chapter** 6 present *PESTT* evaluation tests. Finally, **Chapter** 7, concludes with remarks and insight for future developments.

Chapter 2

The Basis of Software Testing

This chapter introduces some fundamental concepts, like *Test Requirement*, *Coverage*, *Coverage Criterion*, *Graph Coverage*, *Control Flow Graph (CFG)* that are essential to a proper understanding of the project. The introduction of these concepts allow us to establish a parallel between the theoretical concepts and the implemented features presented in the subsequent chapters.

2.1 Fundamental Concepts

There are some ill-defined terms occasionally used in testing, like "complete testing", "exhaustive testing", or "full coverage". These terms are poorly defined because of a fundamental theoretical limitation of software, more specifically, because the number of potential inputs for most programs is so large.

This is where formal coverage criteria come in. Since we cannot test with all inputs, coverage criteria are used to decide which test inputs to use. The software testing community believes that effective use of coverage criteria makes it more likely that test engineers will find faults in a program and provides informal assurance that the software is of high quality and reliability. From a practical perspective, coverage criteria provide useful indicators for when to stop testing. In other words, we can define coverage criteria in terms of test requirements. The basic idea is that we want our set of test cases to enjoy various properties, each of which is provided by an individual test case.

Definition 2.1. ***Test Requirement:*** *A test requirement is a specific element of a software artifact that a test case must satisfy or cover.*

Test requirements usually come in sets and can be described with respect to a variety of software artifacts, including the source code, design components, specification modeling elements, or even descriptions of the input space. For this project we are interested only in the test requirements generated from source code; later in this thesis we explain how to generate them.

Definition 2.2. ***Test Case:*** *A Test case is a set of conditions or variables under which a test engineer will determine whether an application or software system is working correctly or not.*

Suppose now that we have a method with a simple *if-then-else* statement, and we want to cover all decisions. A decision leads to two test requirements, one where the guard evaluates to *true*, which executes the code in the *if* branch, and another when it evaluates to *false*, executing the *else* branch. Calling the method leads to exercise one of this test requirements. A coverage criterion is simply a recipe for generating test requirements in a systematic way.

Definition 2.3. ***Coverage Criterion:*** *A coverage criterion is a rule or collection of rules that impose test requirements on a test set.*

That is, the criterion describes the test requirements in a complete and unambiguous manner.

Test engineers need to know how good a collection of tests is. To do that it is necessary to compare test sets against a criterion in terms of coverage.

Definition 2.4. ***Coverage:*** *Given a set of test requirements* TR *for a coverage criterion* C*, a test set* T *satisfies* C *if, and only if, for every test requirement* tr *in* TR*, at least one test* t *in* T *exists such that* t *satisfies* tr.

Coverage is important for two reasons. First, it is sometimes expensive to satisfy a coverage criterion, so we want to compromise by trying to achieve a certain coverage level.

Definition 2.5. ***Coverage Level:*** *Given a set of test requirements* TR *and a test set* T*, the coverage level is simply the ratio of the number of test requirements satisfied by* T *to the size of* TR.

Second, and more importantly, some requirements cannot be satisfied, making the criterion not 100% satisfied. In this scenario makes all sense to drop unsatisfiable test requirements from *TR*.

Test requirements that cannot be satisfied are called *infeasible*. Formally, no test case values exist that meet the test requirements. Dead code is an example that results in infeasible test requirements because the statements cannot be reached. The detection of infeasible test requirements is formally undecidable for most coverage criteria, and even though some researchers have tried to find partial solutions, they have had only limited success. Thus, 100% coverage is impossible in practice.

Coverage criteria are often related to one another, and compared in terms of *subsumption*. The essence of *subsumption* is: satisfying one criterion guarantees that another one is satisfied.

Definition 2.6. ***Criteria Subsumption:*** *A coverage criterion C_1 subsumes C_2 if, and only if, every test set that satisfies criterion C_1 also satisfies C_2.*

2.2 Graphs

This section starts out with a brief description of graph concepts. Then, continues with the introduction of the coverage criteria applied to graphs used in the project—the Graph Coverage Criteria.

2.2.1 Graph Concepts

Given an artifact under test, the idea is to obtain a graph abstraction of that artifact; in our case the artifact is the source code. The graph abstraction for source code maps code to a Control Flow Graph (CFG). A CFG is a representation, using graph notation, of all paths that might be traversed through a program during its execution. It is important to understand that the graph is not the same as the artifact, and that, indeed, artifacts typically have several useful, but nonetheless quite different, graph abstractions. The same abstraction that produces the graph from the artifact also maps test cases for the artifact to paths in the graph. Formally, a graph is:

- a set *N* of *nodes*.
- a set N_0 of *initial nodes*, where $N_0 \subseteq N$.
- a set N_f of *final nodes*, where $N_f \subseteq N$.
- a set *E* of *edges*, where *E* is a subset of $N \times N$.

Edges are considered to be *from* one node *to* another and are written as (n_i, n_j) (directed edge).

Another important term is *path*. A *path* is a sequence of nodes ([n_1, n_2, ..., n_M]), where each pair of adjacent nodes (n_i, n_{i+1}), $1 \leq i < M$, is in the set *E* of edges. Associated with *path* is the notion of *length*. The *length* of a *path* is defined as the number of edges it contains. A *subpath* of a path *p* is a subsequence of *p* (possibly *p* itself). Like in the edges, we can say that a path is *from* the first node in the path *to* the last node in the path. We can also say that a path is *from* (or *to*) an edge *e*, which simply means that *e* is the first (or the last) edge in the path.

A graph has to contain at least one node in N, N_0, and N_f in order to be useful for generating tests.

Finally, we introduce the term *test path*. A *test path* represents the execution of a test case. A path *p*, possibly of length zero, that starts at some node in N_0 and ends at some node in N_f.

The reason for the *test path* to start at some node in N_0 and to end at some node in N_f is that the execution of a test case always starts at an initial node and ends at a final node.

2.2.2 Graph Coverage Criteria

This section illustrates the different types of coverage criteria based on graphs. Coverage criteria may be divided in two types:

- *Structural Coverage Criteria*;
- *Data Flow Coverage Criteria*.

We proceed by identifying the appropriate test requirements and then define each criterion in terms of the test requirements. In general, for any graph-based coverage criterion, the idea is to identify the test requirements in terms of various structures in the graph. For graphs, coverage criteria define test requirements, *TR*, in terms of the properties of test paths in a graph *G*. A typical test requirement is met by *visiting* a particular node or edge or by *touring* a particular path.

Definition 2.7. ***Visit:*** *A test path* p *is said to visit node* n *if* n *is in* p. *In the same way, test path* p *is said to visit edge* e *if* e *is in* p.

The term *visit* applies well to single nodes and edges, but sometimes we want to turn our attention to subpaths. For subpaths, we use the term *tour*.

Definition 2.8. ***Tour:*** *A test path* p *is said to tour subpath* q *if* q *is a subpath of* p.

The notion of *tour* requires more development. We return to the issue of *touring* later in this section and then refine it further in the context of data flow criteria.

To introduce the graph coverage criteria, we need to adequate the notion of *Coverage* (see Definition 2.4). The following definition is a refinement of that definition:

Definition 2.9. ***Graph Coverage:*** *Given a set* TR *of test requirements for a graph criterion* C*, a test set* T *satisfies* C *on graph* G *if, and only if, for every test requirement* tr *in* TR*, there is at least one test path* p *in* T *such that* p *cover* tr.

This is a very general statement that must be refined for individual cases.

2.2.2.1 Structural Coverage Criteria

We start by defining criteria to *visit* every node and then every edge in a graph. The requirements that are produced by a graph criterion are technically predicates that can have either the value true (the requirement has been met) or false (the requirement has not been met). For *Node Coverage*, we must satisfy a predicate for each node, where the predicate asks whether the node has been *visited* or not. Similarly, for *Edge Coverage*, a predicate must be satisfied for each edge, where the predicate asks whether the edge has been *visited* or not. Formally we have:

Criterion 2.1. ***Node Coverage (NC):*** TR *contains each reachable node in* G.

Criterion 2.2. ***Edge Coverage (EC):*** TR *contains each reachable path of length up to 1, inclusive, in* G.

The test requirements for edge coverage also explicitly include the test requirements for node coverage, that is why the phrase "up to" is included in the definition. In fact, all the graph coverage criteria are developed like this. The motivation is *subsumption* for graphs that do not contain more complex structures.

Other coverage criteria that use only the graph definitions introduced so far, is *Edge-Pair Coverage*

Criterion 2.3. ***Edge-Pair Coverage (EPC):*** TR *contains each reachable path of length up to 2, inclusive, in* G.

Clearly, this idea can be extended to paths of any length, although possibly with diminishing returns. With this context, *Node Coverage* could be redefined to contain each path of length zero, and *Edge Coverage* could be redefined to contain each path of length up to one.

To introduce the next criterion we need a few more definitions. We start with that of *simple path*:

Definition 2.10. ***Simple Path:*** *A path from n_i to n_j is simple if no node appears more than once in the path, with the exception that the first and last nodes may be identical.*

In other words, a simple path have no internal loops, although the entire path itself may wind up being a loop. One useful aspect of simple paths is that any path can be created by composing simple paths. For a coverage criterion for simple paths we would like to avoid enumerating the entire set of simple paths. Since most of the simple paths are subpaths of other simple paths, only maximal length simple paths are considered. Formally:

Definition 2.11. ***Prime Path:*** *A path from n_i to n_j is a* prime path *if it is a simple path and it does not appear as a proper subpath of any other simple path.*

Criterion 2.4. ***Prime Path Coverage (PPC):*** TR *contains each prime path in* G.

Prime Path Coverage has two special cases. Both special cases involve the treatment of loops with "round trips". A round trip path is a prime path of nonzero length that starts and ends at the same node. One type of round trip test coverage requires at least one round trip path to be taken for each node, and another requires all possible round trip paths.

Criterion 2.5. ***Simple Round Trip Coverage (SRTC):*** TR *contains at least one round-trip path for each reachable node in* G *that begins and ends a round-trip path.*

Criterion 2.6. ***Complete Round Trip Coverage (CRTC):*** TR *contains all round-trip paths for each reachable node in* G.

Next we turn to path coverage.

Criterion 2.7. ***Complete Path Coverage (CPC):*** TR *contains all paths in* G.

Complete Path Coverage is not feasible for graphs with cycles, since this results in an infinite number of paths, and hence an infinite number of test requirements.

Touring, Sidetrips, and Detours

We previously defined *"visits"* and *"tours"* (see Definitions 2.7 and 2.8). Recall that using a path p to tour a subpath $[n_1, n_2, ..., n_M]$ means that the subpath is a subpath of p. This is a rather strict, because each node and edge in the subpath must be visited **exactly** in the order that they appear in the subpath. In order to include loops in a tour we can derive the tour definition in two ways:

The first allows the tour to include "sidetrips", where we can leave the path temporarily from a node and then return to the same node.

Definition 2.12. ***Tour with Sidetrips:*** *Test path* p *is said to tour subpath* q *with sidetrips if, and only if, every edge in* q *is also in* p *in the same order.*

The second allows the tour to include more general "detours", where we can leave the path from a node and then return to a forthcoming node on the path (skipping at least one edge).

Definition 2.13. ***Tour with Detours:*** *A test path* p *is said to tour subpath* q *with detours if, and only if, every node in* q *is also in* p *in the same order.*

In the above definitions, q is a required subpath that is assumed to be simple.

2.2.2.2 Data Flow Coverage Criteria

The next few testing criteria are based on the assumption that to test a program adequately, we should focus on the flows of data values. Specifically, we should try to ensure that the values created at one point in the program are used correctly. This is done by focusing on *definitions* and *uses* of values. A *definition (def)* is a location where a value for a variable is stored into memory (assignment, input, etc.). A *use* is a location where a variable's value is accessed. Data flow testing criteria use the fact that values are carried from defs to uses (*du-pairs*). The idea of data flow criteria is to exercise du-pairs in various ways.

First, we must integrate data flow into the existing graph model. Let V be a set of variables that are associated with the program artifact being modeled in the graph. Each node n and edge e is considered to define a subset of V; this set is called *def(n)* or *def(e)*. Each node n and edge e is also considered to use a subset of V; this set is called *use(n)* or *use(e)*.

Before entering the remaining criteria it is necessary to introduce the concept of *def-clear*. A path from location l_i to a location l_j is *def-clear* with respect to variable *v* if for every node n_k and every edge e_k on the path, $k \neq i$ and $k \neq j$, *v* is not in *def(n_k)* or in *def(e_k)*. That is, no location between l_i and l_j changes the value. If a *def-clear* path goes from l_i to l_j with respect to *v*, we say that the def of *v* at l_i reaches the use at l_j. In other words, the variable's value is not changed by another def before it reaches the use.

Another important concept is *du-path*. A *du-path* with respect to a variable *v* is a *simple path* that is *def-clear* with respect to *v* from a node n_i, for which *v* is in *def(n_i)*, to a node n_j, for which *v* is in *use(n_j)*.

The test criteria for data flow are defined as sets of du-paths. This makes the criteria quite simple, but first we need to categorize the du-paths into two groups.

The first grouping of du-paths is according to definitions. Let the *def-path* set *du(n_i, v)* be the set of du-paths with respect to variable *v* that start at node n_i. In other words, all du-paths with respect to a given variable defined in a given node.

The second, and more important, grouping of du-paths is according to pairs of definitions and uses. This is call *def-pair*. Let the *def-pair* set *du(n_i, n_j, v)* be the set of du-paths with respect to variable *v* that start at node n_i and end at node n_j. In other words, all du-paths with respect to a given variable that are defined in one node and used in another (possibly identical) node.

At this point we have the necessary information to present the remaining criteria:

- *All-Defs criterion*;
- *All-Uses criterion*; and
- *All-du-Paths criterion*.

All-Defs criterion requires that each def reaches **at least** one use. The *All-Uses criterion* requires that each def reaches **all possible uses**. Finally, *All-du-Paths criterion* requires that each def reaches all possible uses through **all possible du-paths**. In a formal way:

Criterion 2.8. ***All-Defs Coverage (ADC):*** *For each def-path set* S = *du(n, v),* TR *contains at least one path d in* S.

Criterion 2.9. ***All-Uses Coverage (AUC):*** *For each def-pair set* S = *du(n_i, n_j, v),* TR *contains at least one path d in* S.

Criterion 2.10. ***All-du-Paths Coverage (ADUPC):*** *For each def-pair set* S = *du(n_i, n_j, v),* TR *contains every path d in* S.

Closing the chapter, Figure 2.1 shows the *subsumption* relationship among graph coverage criteria.

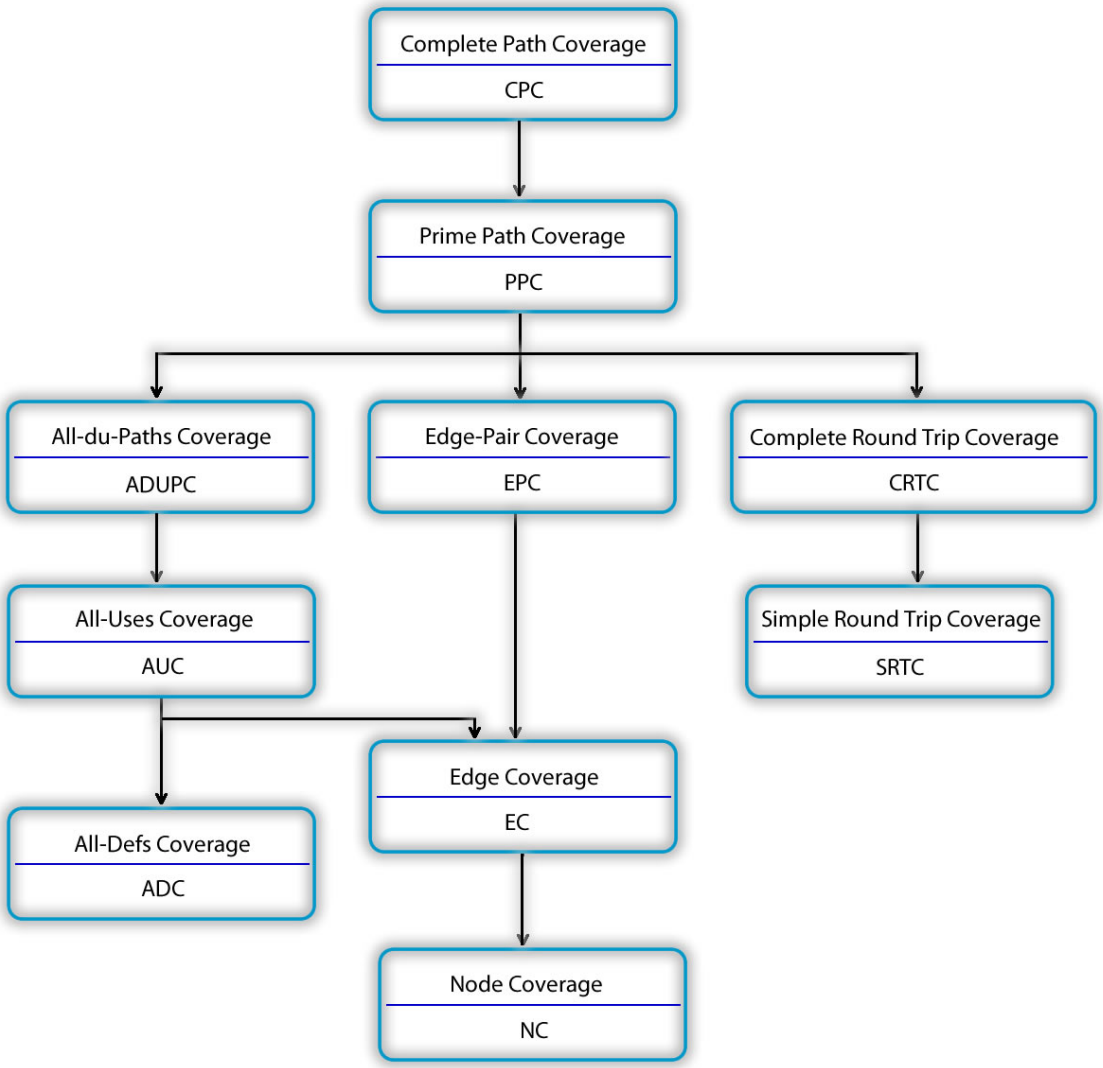


Figure 2.1: The subsumption relationship among graph coverage criteria

Chapter 3

Related Work

There is many available literature that introduces the basic concepts and techniques of *Software* Testing (e.g. [5, 24, 30]), however, there are no tools that support the teaching (and therefore learning) of these techniques and concepts in a comprehensive and integrated way. This chapter presents the related work in the area, indicating some tools and their contributions to this project.

3.1 Visualization Tools

The CFG is a key part in the introduction of software testing concepts. It is therefore essential to provide a good visualization mechanism of the CFG. In this section we present some visualization tools that were tried in the project, as well as a tool that fulfills our requirements.

Graphviz

Graphviz [20] is an open source graph visualization *software*. This tool allows us to represent structural information as diagrams of abstract graphs. To do it *Graphviz* uses many different kinds of languages (depending on the chosen algorithm), such as: *Dot, Fdp, Neato, Twopi, Circo*.

The method of operation is relatively simple: *Graphviz* receives a graph specification and produces a graph description in a simple text language. In order to generate the graph description, *Graphviz* requires a graph specification written in one of its supported languages—in our case the *Dot* language, because we want a directed graph. The graph specification (Figure 3.1a) contains the information about the graph (initial nodes, final nodes, edges, color of nodes, etc), whereas the graph description, in addition to the information provided by the specification, also indicates the exact position of each element (Figure 3.1b).

The *Graphviz* layout programs take the graphs descriptions (in a simple text language), and make diagrams in useful formats, such as images and SVG for web pages, PDF or

```
digraph finite_state_machine {
        rankdir=LR;
        size="8,5"
        node [shape = doublecircle]; LR_0 LR_3 LR_4 LR_8;
        node [shape = circle];
        LR_0 -> LR_2 [ label = "SS(B)" ];
        LR_0 -> LR_1 [ label = "SS(S)" ];
        LR_1 -> LR_3 [ label = "S($end)" ];
        LR_2 -> LR_6 [ label = "SS(b)" ];
        LR_2 -> LR_5 [ label = "SS(a)" ];
        LR_2 -> LR_4 [ label = "S(A)" ];
        LR_5 -> LR_7 [ label = "S(b)" ];
        LR_5 -> LR_5 [ label = "S(a)" ];
        LR_6 -> LR_6 [ label = "S(b)" ];
        LR_6 -> LR_5 [ label = "S(a)" ];
        LR_7 -> LR_8 [ label = "S(b)" ];
        LR_7 -> LR_5 [ label = "S(a)" ];
        LR_8 -> LR_6 [ label = "S(b)" ];
        LR_8 -> LR_5 [ label = "S(a)" ];
}
```

(a) Graph specification.

```
graph 0.70588 11.333 4.9722
node LR_0 0.55556 1.6111 1.1128 1.1128 LR_0 solid doublecircle black lightgrey
node LR_3 5 0.55556 1.1128 1.1128 LR_3 solid doublecircle black lightgrey
node LR_4 5 4.4167 1.1128 1.1128 LR_4 solid doublecircle black lightgrey
node LR_8 10.778 2.2917 1.1128 1.1128 LR_8 solid doublecircle black lightgrey
node LR_2 2.6667 2.2083 1.0017 1.0017 LR_2 solid circle black lightgrey
node LR_1 2.6667 0.81944 1.0017 1.0017 LR_1 solid circle black lightgrey
node LR_6 5 2.6111 1.0017 1.0017 LR_6 solid circle black lightgrey
node LR_5 6.9444 1.6111 1.0017 1.0017 LR_5 solid circle black lightgrey
node LR_7 8.8333 1.2917 1.0017 1.0017 LR_7 solid circle black lightgrey
edge LR_0 LR_2 4 1.0937 1.7634 1.3851 1.8458 1.7462 1.948 2.049 2.0336
"SS(B)" 1.6389 2.0972 solid black
edge LR_0 LR_1 4 1.0774 1.4154 1.3763 1.3033 1.7526 1.1622 2.0641 1.0454 "SS(S)"
1.6389 1.4167 solid black
edge LR_8 LR_6 10 10.226 2.3881 9.5663 2.498 8.4289 2.6708 7.4444 2.7361 7.001
2.7655 6.8885 2.7553 6.4444 2.7361 6.1804 2.7247 5.8901 2.7016 5.6393 2.6784
"S(b)" 7.8889 2.8333 solid black
edge LR_8 LR_5 7 10.223 2.2201 9.6031 2.1359 8.5713 1.9829 7.6944 1.7917 7.6532
1.7827 7.6108 1.7732 7.5681 1.7634 "S(a)" 8.8333 2.2083 solid black
```

(b) Graph description.

Figure 3.1: Graph description in a simple text language for a given specification in *Dot*

Postscript for inclusion in other documents; or display in an interactive graph browser. Also, *Graphviz* has many useful features for concrete diagrams, such as options for colors, fonts, tabular node layouts, line styles, hyperlinks, rolland custom shapes.

The Figure 3.2 shows a CFG obtained with *Graphviz*.

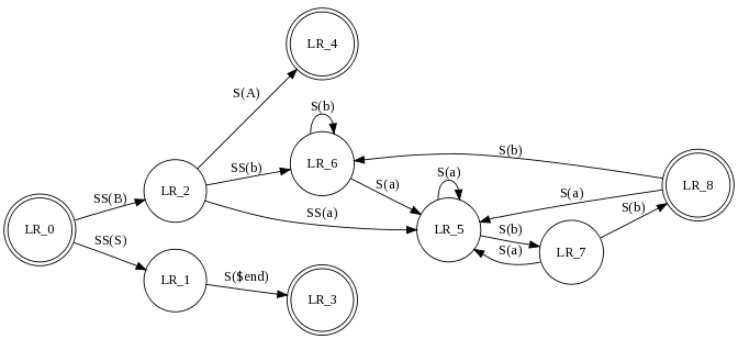


Figure 3.2: The CFG for the specification shown in Figure 3.1a

This tool was an obvious choice to be integrate in this project for the following reasons:

- allows for drawing high quality CFGs (intersections rarely occur);
- the amount and variety of available options;
- open source tool—its license allows the inclusion in a third party software with non-commercial purposes.

Zest

Although *Graphviz* is an excellent tool, it does not offer the required integration with *Eclipse* needed for the project, in other words, the CFGs produced by *Graphviz* are static and do not allow any type of interactivity in *Eclipse* making them useless directly.

Fortunately, we found a tool that may be used to integrate *Graphviz* in *Eclipse*, *Zest* [37]. *Zest* is a visualization toolkit plug-in for *Eclipse* that has been developed in *SWT/Draw2D* and integrates seamlessly within *Eclipse* because of its recognized design. It has a predefined set of classes, interfaces, and operations to help building graphics on a *Graphical Editing Framework (GEF)* [18] editor. It has drag and drop support for its elements, uses animations when the graph is being drawn, and, mainly, allows an easy interaction between the graph and the remaining elements of *Eclipse*. *Zest* also offers some default layout algorithms; namely:

- Spring Layout Algorithm;
- Fade Layout Algorithm;
- Tree Layout Algorithm;
- Radial Layout Algorithm;
- Grid Layout Algorithm.

However, *Zest* presents some limitations, the most significant are: none of the default algorithms are powerful enough to display a CFG, and the visual appearance of the displayed graph is far from acceptable. These constraints mean that for the same graph specification *Zest* and *Graphviz* generate two very different graphs. In the project these limitations have a significant impact, since a good graph representation is essential.

To build the CFG, we implemented a new layout algorithm for *Zest* that integrates the graph produced by *Graphviz* into *Zest*, which we explain in detail in Section 5.1.3.

Eclipse Control Flow Graph Generation

The *Eclipse Control Flow Graph Generation* [19] is a plug-in for *Eclipse* that generates control flow graphs for *Java* code. Internally, it uses a graph model based on a toolkit called *Java Development Toolkit (JDT)* to handle operations with *Java* code. To do so, they traverse an *Abstract Syntax Tree (AST)* created by *JDT* using a visitor (*ASTVisitor*), associating each *Java* statement to a node in the graph model. Internally, they modified the *Zest* tree algorithm to display the graph model, as a CFG. For coverage purpose, this tool interacts with *CodeCover* [9] plug-in for *Eclipse* that we detail in Section 3.2.2.

Although it generates the graphs based on the evaluation of the source code and provides some coverage information with the help of *CodeCover*, *Eclipse Control Flow Graph Generation* does not meet our needs because:

- the generated CFG is not suitable for applying the coverage criteria specified in Section 2.2.2;
- user interaction is very limited, as well as the information available;
- third party dependency for coverage, making limited the available coverage criteria. Its expansion would make the process too complex, further increasing the dependency.

3.2 Generators and Recognizers

In practical terms, we have two types of commercial automated testing tools generators and recognizers. A generator corresponds to a tool that automatically creates test case values; a recognizer is a coverage analysis tool. There is also a third group containing the tools that automatically create test scripts, however, this group of tools will not be addressed in this thesis, because the automatic creation of test scripts is not part of our goals.

Some of the available generators are "part" of *xUnit*. *xUnit* is the generic name by which the various code-driven testing frameworks are collectively known. These frameworks allow testing of different elements (*units*) of *software*, such as methods and classes. The main advantage of *xUnit* frameworks is that they provide an automated solution with no need to write the same tests many times, and no need to remember what should be the result of each test. Such frameworks are based on a design made by *Kent Beck* [4], originally implemented for *Smalltalk* [34] as *SUnit* [35].

Generator and recognizer tools are quite plentiful, both as commercial products and freeware.

3.2.1 Generators

Some of the most popular generators for *Java* language are:

JUnit

JUnit [26] is an open source unit testing framework that has been designed by *Erich Gamma* [16] and *Kent Beck* (ported from *SUnit*) for the purpose of writing and running tests in the *Java* programming language. It was the base of the *xUnit* architecture for unit testing frameworks that has been used in many programming languages, such as *C*, *C#*, and *PHP*, giving rise to various unit testing tools.

There are many ways to write test cases. A test case defines the tuple of input values, the expected values, and other information needed. These test cases need to be converted into executable fragments—test script. A test script is a test automation program code written in a programming language that checks that another code unit works as expected. So, if we want an accurate and efficient testing process, then using a good testing framework is recommended. *JUnit* has established a good reputation in this scenario.

Using a framework, like *JUnit*, to develop test cases has a number of advantages, the most important being that others are able to understand test cases, write new ones, and that most development tools enable for automated or one click test case execution.

JUnit also provides a graphical user interface (GUI)–only in frameworks like *Eclipse*– that makes it possible to write and test source code quickly and easily. *JUnit* shows test progress in a bar that is green if testing are going fine and it turns red when a test fails. There is a lot of pleasure in seeing the green bar growing in the GUI output. A list of unsuccessful tests appears at the bottom of the display window. We can run multiple tests concurrently. The simplicity of *JUnit* makes it possible for the *software* developer to correct bugs as they are found.

TestNG

TestNG [36] is a general purpose open source unit testing framework inspired from *JUnit* and *NUnit* [29], but introducing some new functionalities in order to make it more powerful and easier to use. *TestNG* is available for *Eclipse* as a plug-in and is also one of the *xUnit* frameworks. Some of the features offered by *TestNG* are: flexible test configuration, support for data-driven testing (with @DataProvider), support for multiple instances of the same test class (with @Factory), support for parameters, allows for distribution of tests on worker machines, and allows for generation of test reports in HTML and XML formats.

3.2.2 Recognizers

Most recognizers interact internally with a generator in order to function properly. In other words, recognizers use generators to provide coverage information about executed test cases. Some of the most popular recognizers for the *Java* language are:

Jtest

Jtest [25] is a commercial integrated solution for automating a broad range of practices proven to improve development team productivity and software quality. It focuses on practices for validating *Java* code and applications. *Jtest* is one of the *xUnit* frameworks, meaning that it can be considered also as a generator. *Jtest* facilitates unit testing, static analysis, peer code review, and runtime error detection.

As *Jtest*, *PESTT* also perform code static analysis, however, in *PESTT* peer code review and runtime error detection are not supported, as it is not part of our immediate goals. Likewise, *PESTT* offers CFG visualization and additional coverage criteria not available in *Jtest*.

Google CodePro Analytix

Google CodePro Analytix [1] is an open source coverage analysis tool. However, this tool is much more complete than the other coverage analysis tools, because it offers many features, such as code analysis, many different types of metrics, *JUnit* test generation, *JUnit* test editor, code coverage, dependency analysis, similar code analysis, and Javadoc maintenance.

Google CodePro Analytix is the premier *Java software* testing tool for *Eclipse* developers who are concerned about improving *software* quality and reducing development costs and schedules. It provides some of the features that are also available in by *PESTT*, such as code coverage and some of the metrics. However, in code coverage *PESTT* provides a larger number of criteria, beyond the coverage criteria based on nodes and edges available in *Google CodePro Analytix*. It also provides automatically *JUnit* test generation, dependency analysis, similar code analysis, and Javadoc maintenance, features that are outside the scope of *PESTT*. About teaching software testing techniques, *Google CodePro Analytix* is not very useful, since the features offered are geared for those who already know what to test and how to do so, and not for beginners.

Clover

Clover [7] provides metrics for better balancing the effort between writing code that does stuff and code that tests stuff. *Clover* runs in *Eclipse* and *IntelliJ* IDE or in continuous integration system, and includes test optimization to make tests run faster and fail more quickly. *Clover* is a commercial product, but it is freely available to open source projects and non-profit institutions. Some of the features available are: color-based visual annotation of test coverage, line-by-line test coverage info in source files, pass/fail information for recent test runs.

As *Clover*, *PESTT* also include color-based visual annotation of test coverage and line-by-line test coverage info in source files. *PESTT* does not perform any task regarding test writing or test execution; this is done internally by *JUnit*. *Clover* has its own mechanisms for test optimization and test runs.

codeCover

codeCover is a free glass-box testing tool. *codeCover* measures statement, branch, loop, and term coverage (subsumes Modified Condition/Decision Coverage (MC/DC)), ques-

tion mark operator coverage, and synchronized coverage (used to synchronize critical code sections. If a synchronized statement is locked, other threads will wait until the locked section is set free. The synchronized coverage metric shows whether a synchronize statement has caused waiting effects.). It allows report creation (and customization) in HTML. *codeCover* can be integrated in *Eclipse*, *Batch*, and *Ant*. It also provides color-based visual annotation of test coverage, line-by-line test coverage info in source files, pass/fail information for recent test runs, and coverage statistics.

codeCover provides features that are not supported by *PESTT*, such as: term coverage, synchronized coverage, or report generation. Likewise, *PESTT* provides a CFG relative to the source code and some other coverage criteria like *Prime path* coverage, *All-du paths* coverage, etc.

EclEmma

EclEmma [12] is an open source *Eclipse* plug-in based on *JaCoCo* [23], inspired in *Emma* [13]. It uses the information generated from *JUnit* test run. *EclEmma* offers features like coverage overview details, source code highlighting, different types of metrics (instructions, branches, lines, methods, types, or cyclomatic complexity), multiple coverage sessions, merge sessions, and import/export features (execution data import and coverage report export).

PESTT provides most of the features available in *EclEmma*, coverage overview details, source code highlighting, different types of metrics (instructions, branches, lines, methods, types or cyclomatic complexity), besides more coverage criteria and a CFG viewer. In the other hand, *PESTT* do not offer report generation or operations over sessions, which are provided by *EclEmma*.

There are many other coverage analysis tools for the *Java* programming language like *Cobertura* [8], *DevPartner* [11] and *JMockit Coverage* [10], but any of them does what we propose.

3.2.3 Instrumentation

An important concept related to coverage analyzers is code instrumentation. Coverage analyzers work by adding information to the original source code—code instrumentation. For *Java*, coverage analyzers fall into three categories:

- those that run the code in a modified *Java Virtual Machine (JVM)*;
- those that add instrumentation to the *Java* bytecode; and
- those that insert instrumentation into the *Java* source code.

The following table compares the different methods of obtaining code coverage and their relative benefits:

Possible Features	***JVM***	**Bytecode**	**Source code**
Can work without source	yes	yes	no
Compilation time	no impact	variable	variable
Container friendly	no	no	yes
Control which entities are reported on	limited	limited	yes
Gathers branch coverage	indirectly	indirectly	yes
Gathers method coverage	yes	yes	yes
Gathers source metrics	no	no	yes
Gathers statement coverage	line only	indirectly	yes
Requires separate build	no	no	yes
Requires specialized runtime	yes	yes	no
Runtime performance	high impact	variable	variable
Source level directives to control coverage gathering	no	no	yes
View coverage data in-line with source	not accurate	not accurate	yes

Table 3.1: Comparison of three types of instrumentation in *Java* (Clover [22])

We chose to add instrumentation directly to the bytecode, because we feel this is the best approach, since it does not require a modified VM, but still retains a big speed advantage over having to compile all the source code twice.

Next, we present two tools that instrument the *Java* bytecode: *JaCoCo* and *Byteman* [6].

JaCoCo

JaCoCo is a free *Java* code coverage library distributed under the *Eclipse Public License* [27]. The *JaCoCo mission* is to become the standard backend technology for code coverage analysis in JVM based environments. Their focus is on providing a lightweight, flexible and well-documented library for integration with various build and development tools.

JaCoCo uses a set of different counters to compute coverage metrics. All these counters are derived from information contained in *Java* class files, which basically are *Java* bytecode instructions and debug information optionally embedded in class files. *JaCoCo* provides the following metrics: instructions coverage, branches coverage, cyclomatic complexity, line coverage, method coverage, and classes coverage.

To provide the above features, coverage information has to be collected at runtime. For this purpose *JaCoCo* creates instrumented versions of the original class definitions. The instrumentation process happens on-the-fly during class loading using *Java agents*. The *Java agents* are loaded by the application class loader, therefore, the classes of the

agent live in the same name space as the application classes. The *JaCoCo* build, moves all agent classes into a unique package. Instrumentation requires mechanisms to modify and generate *Java* bytecode. *JaCoCo* uses the *ASM* [3] library for this purpose, internally.

The strategy used by *JaCoCo* for instrumentation is to insert probes into the control flow at runtime and analyze the actual code coverage. Probes are additional instructions that can be inserted between existing instructions. They do not change the original behavior of the method, but record the fact that they have been executed. *JaCoCo* implements probes with a `boolean` array instance per class. Each probe corresponds to am entry in this array. Whenever the probe is executed, the entry is set to *true*. With this strategy it knows exactly the nodes that were executed.

Now, we illustrate the *JaCoCo* mode of operation, through a simple example, based on the following method:

```
1  public class DemoClass {
2
3    public void demo() {
4      for(int x = 0; x <= 3; x++) {
5        if(x % 2 == 0)
6          System.out.println("even");
7        else
8          System.out.println("odd");
9        if(x >= 2)
10         System.out.println("greater or equal than 2");
11       else
12         System.out.println("less than 2");
13     }
14   }
15 }
```

Listing 3.1: Method used to explain the operation of *JaCoCo* and *Byteman*.

The previous method is extremely simple: a variable$x$ varies between $[0..3]$, and each loop iteration verifies if the current value of x is even or odd (the first if-else block) and, after, if x is greater or equal than 2 (the second if-else block). The goal is to illustrate the analisys of the so called *double diamond* construction. This example uses a double diamond inside a for loop, in order to make it slightly more interesting. For a better understanding of the method, Figure 3.3 shows the corresponding CFG. The correspondence between the nodes of the CFG and the method is done as follows:

- Node 0 - `int x = 0;`
- Node 1 - the begin of `for` statement;
- Node 2 - does not have any instruction (corresponds to an implicit `return`);
- Node 3 - the begin of the first `if` statement;
- Node 4 - `System.out.println("even");`
- Node 5 - the begin of the second `if` statement;

- Node 6 - `System.out.println("greater or equal than 2");`
- Node 7 - `x++;`
- Node 8 - `System.out.println("less than 2");`
- Node 9 - `System.out.println("odd");`

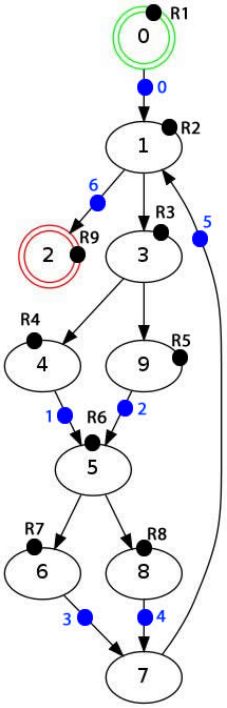


Figure 3.3: The CFG for Listing 3.1

To test our example we use the following test script:

```
1 public class TestDemoClass {
2
3   @Test
4   public void testDemo() {
5     DemoClass dc = new DemoClass();
6     dc.demo();
7   }
8 }
```

Listing 3.2: A test for the method shown in Listing 3.1

As a probe implementation itself requires multiple bytecode instructions this would increase the size of the class files and slowdown execution speed of the instrumented classes. In order to reduce these risks, *JaCoCo* uses a few number of probes, inserting them in particular places. It inserts probes at every:

- method exit (return or throws);
- edge where the target instruction is the target of more than one edge.

If a probe has been executed, *JaCoCo* knows that the corresponding edge has been visited. From this edge it can infer about other preceding nodes and edges that:

- if an edge has been visited, it knows that the source node of this edge has been executed;
- if a node has been executed and the node is the target of only one edge it knows that this edge has been visited.

The blue marks shown in the Figure 3.3 illustrates the places where the probes are inserted by *JaCoCo* to instrument the method shown in Listing 3.1. With this probes *JaCoCo* can collect the necessary information to generate node and edge coverage.

When running the test (Listing 3.2) *JaCoCo* starts with a `boolean` array of size seven with its elements set to *false*. As Figure 3.3 shows each probe has a number that corresponds to the index in the array. In the first loop iteration probes 0, 1, 4, and 5 are executed, setting the corresponding array elements to *true*. The second iteration executes probes 2, 4, and 5. Probes 4 and 5 were already executed in loop's first iteration, so the values in the array for these elements are already set to *true*. The third loop iteration executes probes 1, 3, and 5, which only sets the array elements corresponding to probe 3 to *true*. The fourth loop iteration executes probes 2, 3, and 5. The array does not change because these elements are already set. When the method exits probe 6 is executed, setting the corresponding array element to *true*.

At the end of the method execution all array elements are *true*. With this information *JaCoCo* guesses node and edge coverage. To compute the final node coverage result *JaCoCo* needs to interpret the collected information. Probe 0 visited the edge (0, 1) so it executes nodes 0 and 1. Probe 1 visited edge (4, 5), meaning that nodes 4 and 5 are executed. As we indicated above, if a node has been executed and it is the target of only one edge, *JaCoCo* infers that this edge has been visited, which means that edge (3, 4) is visited and so node 3 is executed. Applying this rule to probes 2, 3, and 4, it concludes that nodes 6, 7, 8, and 9 are also executed (excluding those already executed). Probe 5 does not add new nodes to the list of executed nodes (probe 5 visits edge (7, 1), meaning nodes 1 and 7 executes). Finally, probe 6 indicates the execution of node 2. As final result we get that all nodes are covered. With only the information stored in the `boolean` array, *JaCoCo* cannot determined *Edge-pair* coverage, *Prime Path* coverage etc. Since it requires to know if an edge is visited after another edge has been visited just before.

Byteman

Byteman is a tool that simplifies tracing and testing of *Java* programs. *Byteman* allows us to insert extra *Java* code into an application, either as it is loaded during JVM startup or even after it has already started running. The injected code is allowed to access any application's data and call any application's method. We can inject code almost anywhere and

there is no need to prepare the original source code in advance, nor the need to recompile, repackage, or redeploy the application. In fact, we can remove injected code and reinstall different code while the application continues to execute.

Byteman works by modifying the bytecode of the application classes at runtime. Since it only needs access to bytecode, this means it can modify library code whose source is either unavailable or unable to be recompiled.

Byteman uses a clear, simple scripting language, based on a formalism called *Event Condition Action (ECA)* rules to specify where, when, and how the original *Java* code should be transformed. An event specifies a trigger point, a location where we want code to be injected. When execution reaches the trigger point, the rule's condition, a *Java* boolean expression, is evaluated. The *Java* expression (or sequence of expressions) in the rule action is executed only when the condition evaluates to *true*. Normally, execution continues from the trigger point once the injected code has been executed. However, rule actions may also throw an exception or force an early return from the triggering method.

Now, we illustrate the *Byteman* mode of operation, through a simple example, based on the method shown in 3.1.

The first thing to do is to define the rules to be used; so, we need to specify where, when, and how the original *Java* code should be transformed. We do not want to change the original *Java* code, just need to be notified when a CFG node is executed. The simplest approach would be to define a rule for each CFG node. We know that one node is executed if the program executes the first instruction associated with it, so placing a trigger point at the line of the first instruction of each node allows us to know if the node is executed (depending on whether the rule is executed or not). Listing 3.3 shows the rules to get node coverage through *Byteman*.

```
1  RULE r2
2  CLASS DemoClass
3  METHOD demo
4  AT LINE 4
5  IF true
6  DO System.out.println("1 7")
7  ENDRULE
8
9  RULE r3
10 CLASS DemoClass
11 METHOD demo
12 AT LINE 5
13 IF true
14 DO System.out.println("3")
15 ENDRULE
16
17 RULE r4
18 CLASS DemoClass
19 METHOD demo
```

```
20 AT LINE 6
21 IF true
22 DO System.out.println("4")
23 ENDRULE
```

Listing 3.3: Rules to get node coverage through *Byteman*.

The entire list of rules can be seen in Appendix A (Section A.1).

Each rule is defined by a set of elements:[1]

- RULE - the name of the rule (must be unique or *Byteman* will only run the last rule with this name); also indicates the beginning of a new rule;
- CLASS - the name of the class where the rule will be inserted;
- METHOD - the name of the method where the rule will be inserted;
- AT LINE - the line number where the rule will be inserted;
- IF - the evaluation condition;
- DO - the code to be executed if the condition evaluates to *true*;
- ENDRULE - indicates the end of the rule.

Of the elements listed above the only one which is not required is AT LINE, all the others are part of any rule definition. In this case the use of AT LINE makes all sense, because we know, if the program executes an instruction in a particular line, and this line has associated a trigger point, then we are sure that the node will be executed (the reason why the evaluation condition is just *true*). Since the evaluation condition is *true*, every time a trigger point is reached, the condition is executed, in other words, the number of the node is printed.

As we did for *JaCoCo*, we illustrate *Byteman*'s mode of operation to get node coverage. Running the test shown in Listing 3.2 generates the following output:

```
1  0
2  1 7
3  3
4  4
5  pair
6  5
7  8
8  less than 2
9  3
10 9
11 odd
12 5
13 8
14 less than 2
15 3
16 4
```

[1] These are some of the available elements that can be used in *Byteman* rule's definition.

```
17 pair
18 5
19 6
20 greater or equal than 2
21 3
22 9
23 odd
24 5
25 6
26 greater or equal than 2
27 2
```

Listing 3.4: Output generated by running the test (Listing 3.2) with the *Byteman* rules.

The output generated may seem a little strange at first glance. Probably, the reader would expect more lines with 1 7 (at least one for each loop iteration). The truth is that the rule is trigger just before reaching the beginning of the loop. For a better understanding, look for the black marks shown in Figure 3.3. Each rule is placed on the node's extremity (immediately before executing the first instruction of the node). Let us clarify this situation by going through the output. Once the test invokes the method, rule *r1* is executed, printing 0. As the program execution is sequential, rule *r2* is executed, printing 1 7. Then, prints 3 and 4 (the result of executing rules *r3* and *r4*) and "even" (by executing `System.out.println("even")`). This happens because rule *r4* is triggered just before the program reaches line 6 (where the instruction `System.out.pri- ntln("even")` is located). Continuing program execution, 5, 8 and "less than 2" are printed corresponding to the execution of rules *r6*, *r8*, and the `System.out.println("less than 2")` instruction. At this point, the output generated fits perfectly with what is expected. Now, comes the tricky part. Node 8 (corresponding to `System.out.println("less than 2")`) is connected to node 7 (corresponding to `x++`). Inspecting Listing 3.3, node 7 is associated with the rule *r2* (placed in line 4). Although, instructions at line 4 (`x <= 3` and `x++`) are executed several times (because of loop iterations), rule *r2* is executed only once. Recall that rule *r4* is triggered just before the program reaches line 4. The only way to reach line 4 is coming from line 3, the reason why 1 and 7 appear just once. The rest of the program is similar to that already explained. When the program ends rule r9 is executed, printing 2 tat corresponds to the final node. Looking at the output, we can see that all nodes were printed. However, it does not mean that all nodes have been covered. To make sure of that we need to look at the rules that print more than one node, more specifically at rule *r2* that prints nodes 1 and 7. Node 1 is executed because line 4 is reached, but we do not know if node 7 is executed. We only know that node 7 is executed if `x <= 3` evaluates to *true*, meaning that the program execution enters the loop's body. In the present case we know this had happened because node 3 is printed. So, `x++` is executed, hence node 7 is executed. As with *JaCoCo*, with *Byteman* we get that all nodes are covered (as expected), even with a bit more work.

JaCoCo vs. *Byteman*

We shown the mode of operation of *JaCoCo* and *Byteman* above, although the example used is very simple, it allowed us to identify some limitations in view of the needs required for the *PESTT*. Let now compare the two tools.

The both tools are quite similar with respect to obtaining the node coverage (as well as edge coverage), because *Byteman* can replicate the operation of the *JaCoCo* probes—through the rules. The main difference between the two tools is the flexibility. *JaCoCo* is not very flexible. It can only provide coverage results based on the information stored in the `boolean` array (corresponding to the execution of probes). With only this information available we cannot do much more than what the *JaCoCo* already offers—node and edge coverage. Another negative aspect is that it does not allow us to define our own probes, in order to get more information. Instead, *Byteman* offers a more wider and flexible way of obtaining information, by setting our own rules, placing them and define the behavior we want. For example, with *JaCoCo* we can only know if a node is executed or not, but we cannot know how many times each one is executed (one of the reasons why *JaCoCo* do not offer other "more powerful" coverage criteria). In the opposite way, with *Byteman* we can obtain much more information (depending on how the rules are defined), such as the number of times each node is executed or the sequence in which they are executed.

Having made this comparison we can conclude that *Byteman* is the ideal tool for us to use, not only by offering us everything *JaCoCo* already offers, but it allows us to obtain the necessary information so we can provide all coverage criteria based on graphs as specified in Section 2.2.2.

3.2.4 Contributions

Despite the variety of available coverage tools, none of them provides coverage based on CFGs (one of the reasons that led to this project). Our goal is not to develop a software testing tool, but to support the teaching of the techniques and concepts used in these tools, and, therefore, the writing of sets of test cases and the evaluation of their coverage results.

Like most of the coverage analysis tools mentioned before, we also need a generator for writing and running the test cases, in order to get the execution paths in the CFG of the run tests.

As generator we choose *JUnit* for the following reasons:

- it is open source;
- it is already integrated with *Eclipse*;
- it has an excellent API; and, foremost,
- it does exactly what we need in a simple and fast way.

Although, we do not use any coverage analysis tool in *PESTT*, the assessment of these tools allowed us to identify some very useful features, which we can highlight:

- line-by-line, color-based visual test coverage;
- graphical statistics of coverage;
- launch *JUnit* as part of our plug-in;
- coverage reports.

As any coverage analysis tools, we also need to instrument code to gather information, so that we can provide coverage metrics. For this purpose, we choose *Byteman* because:

- it is open source;
- it has an excellent API;
- it is quite flexible;
- it can be used in conjunction with *JUnit*; and, foremost,
- it allow us to do what we want in a simple and fast way.

Chapter 4

Architecture

This chapter gives an overview of the *PESTT* architecture. We start by describing the *Eclipse* architecture, giving an overview of its internal organization and its major components. Then, we describe *PESTT*'s architecture presented in two parts: the first concerning the integration with *Eclipse* and the necessary dependencies for its operation; the second is related to the internal organization of the *PESTT* plug-in, where we explain in detail the design decisions taken during the development process and other relevant aspects.

4.1 Overview

Eclipse is a multi-language software development environment comprising an integrated development environment (IDE) and an extensible plug-in system. It is written mostly in *Java*. It can be used to develop applications in *Java* and, by means of various plug-ins, in many other programming languages. Users can extend its abilities by installing plug-ins written for the *Eclipse Platform*, such as development toolkits for other programming languages, and can write and contribute with their own plug-in modules.

4.1.1 *Eclipse* Architecture

Eclipse is a complex application, but at the same time has a simple to understand architecture. It is built on its own implementation of the *Open System Gateway Initiative (OSGi)* [31] notion, called Equinox [2, 15]. Some of the reasons why the *Eclipse* team choose to use an infrastructure based on *OSGi* are:

- named, versioned bundles;
- dependency management;
- explicit imports/exports;
- built-in security;
- brings modularity to *Java*;
- independent industry standard.

The *OSGI* architecture is pictured below:

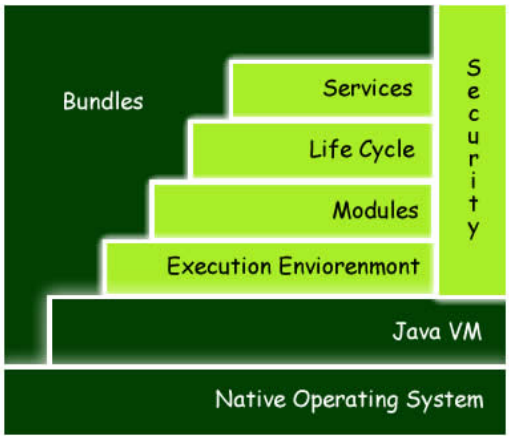


Figure 4.1: The *OSGi* architecture [31]

Equinox is a module runtime that allows developers to implement an application as a set of bundles (in the OSGi notation or plug-ins as they are called on *Eclipse*) using common services and infrastructure.

The *Eclipse* platform consists of an architecture built entirely around plug-ins. Plug-ins are structured bundles of code or data that contribute to the system with functionalities, in the form of code libraries (*Java* classes with public API), platform extensions, or even documentation. Plug-ins can define extension points, well-defined places where other plug-ins can add functionality. All plug-ins integrate with *Eclipse* platform in exactly the same way. Each subsystem in the platform is itself structured as a set of plug-ins that implement some key functions. Some plug-ins add visible features to the platform using the extension model. Others supply class libraries that can be used to implement system extensions. More concretely, a plug-in minimally consists in a bundle manifest file, `MANIFEST.MF`. This manifest provides details about the plug-in, such as its name, ID, and version number. The manifest may also tell the platform what *Java* code it supplies and what plug-ins it requires, if any. Note that everything except the basic plug-in description is optional. A plug-in typically also provides a plug-in manifest file, `plugin.xml`, that describes how it extends other plug-ins, or what capabilities it exposes to be extended by others (extensions and extension points). This manifest file is written in *XML* and is parsed by the platform when the plug-in is loaded. All the information needed to display the plug-in in the UI, such as icons, menu items, and so on, is contained in the manifest file.

The *Eclipse* Platform itself just provides the environment for the plug-ins to operate. The basic platform consists of the following components:

- the Platform runtime;
- the Workspace that “holds” the development environment;

- the Workbench, which implements the graphical interface to *Eclipse*, and its sub-components JFace and the Standard Widget Toolkit (SWT);
- the Version and Configuration Management system (VCM);
- the Help system.

The *Eclipse* architecture is depicted below:

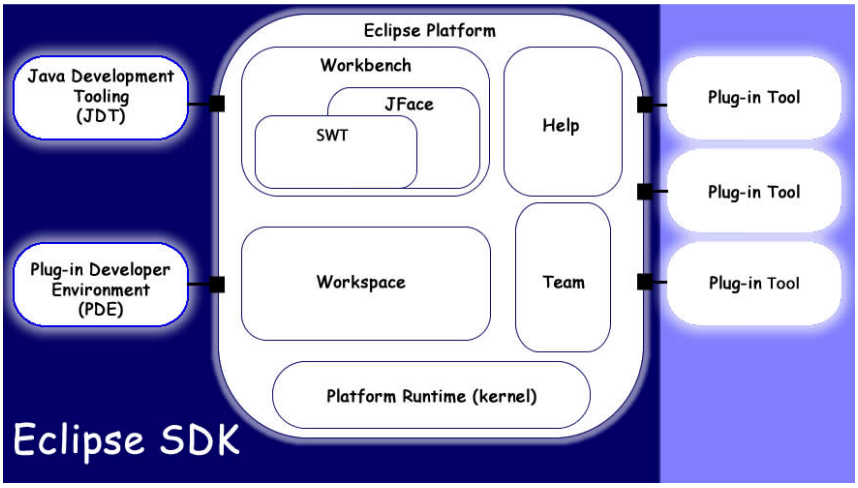


Figure 4.2: The *Eclipse* architecture [21]

The Platform runtime is an essential core of *Eclipse* that performs the task of creating and managing the *Java* execution environment for the development tool. It loads and ties together the other basic plug-ins. With the exception of the runtime kernel, everything else in *Eclipse* is a plug-in. The Workbench component contains extension points that, for example, allows a plug-in to extend the *Eclipse* user interface with menu selections and toolbar buttons, to request notification of different types of events, and to create new views. The Workspace component contains extension points that allow plug-ins to interact with resources, including projects and files. There is also a Team component that allows *Eclipse* resources to interact with version control systems (like VCS or Git), but unless building an *Eclipse* client for a VCS, the Team component will not have its functionality extended or enhanced. Finally, there is a Help component available to provide online documentation and context-sensitive help for applications.

The *Eclipse* SDK also includes two major tools that are useful for plug-in development in addition to the basic platform. The *Java Development Toolkit (JDT)* implements a full featured *Java* development environment. The Plug-in Developer Environment (PDE) adds specialized tools that streamline the development of plug-ins and extensions.

Eclipse uses a load-on-demand strategy, which makes it feasible to have many different plug-ins and still have reasonable performance.

4.2 Integration and Dependencies

As mentioned in the previous section, the *Eclipse* Platform consists of an architecture built entirely around plug-ins, therefore, not surprisingly, that the integration of *PESTT* in *Eclipse* is done through a plug-in.

In order to facilitate the creation and subsequent integration of plug-ins, *Eclipse* provides a wizard with a set of templates. Through these templates we can start creating the *PESTT* plug-in from scratch [33]. After we identified the required dependencies to interact with *Eclipse* and understood how they can be extended and how add new dependencies to our plug-in. We identified *Zest* as a critical dependence. As mentioned in Section 3.1, we intended to use *Zest* to design and visualize CFGs in an integrated and interactive way. Since *Zest* is also an *Eclipse* plug-in, its integration was straightforward: just add a dependency to it in *PESTT*'s bundle manifest file. This means that the external plug-in (*Zest*) contributes features to *PESTT* and that it is required to be on the project's classpath in order to compile.

Due to *Zest* limitations (described in Section 3.1) it was necessary to add an external dependency to *Graphviz*, so that we can show a perceptible CFG. However, this dependence is different, since *Graphviz* is not an *Eclipse* plug-in. We explain how we interact with *Graphviz* in Section 5.1.3.

Another dependency is *JUnit*. The goal of integrating *JUnit* into our plug-in is to control the execution of tests to obtain coverage information, more precisely, run tests, collect data, and show the execution results to the user, all within the plug-in. Section 5.3.2 details the integration with *JUnit*. Since *JUnit* is part of *Eclipse* JDT as a built in plug-in its integration was straightforward.

An exact list of *PESTT* dependencies together with their versions are shown below:

```
1 org.eclipse.ui;bundle-version="3.7.0",
2 org.eclipse.zest.dot.core;bundle-version="2.0.0",
3 org.eclipse.ui.console;bundle-version="3.5.100",
4 org.eclipse.jdt.launching;bundle-version="3.6.0",
5 org.eclipse.debug.ui,
6 org.eclipse.jdt.junit;bundle-version="3.7.0"
```

Listing 4.1: *PESTT* list of dependencies.

The entire `MANIFEST.MF` file is provided in Appendix A (Listing A.2). Plug-in dependencies are illustrated in the Figure 4.3.

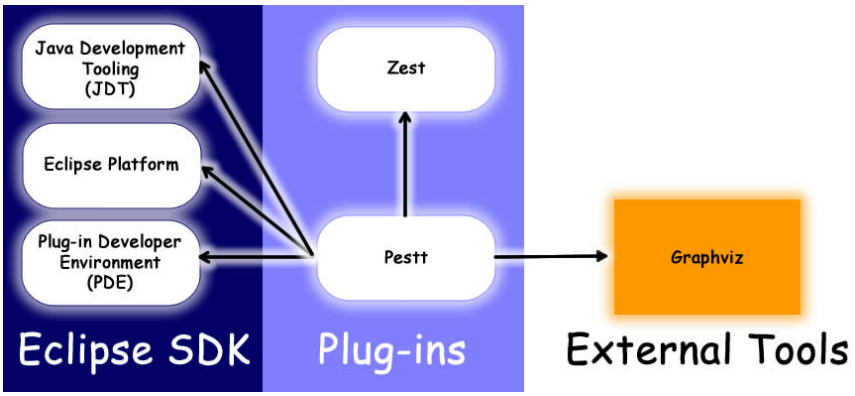


Figure 4.3: Simplified view of *PESTT* dependencies

4.3 Internals

PESTT, can be divided into 4 major parts from the bundle point of view:

- main;
- adts;
- domain;
- ui.

The *main* part activates the plug-in. It contains the plug-in activator, which is *PESTT* connection with the outside environment, meaning that it is the plug-in interface for *Eclipse*. The plug-in activator loads the necessary structures for the proper functioning of the plug-in. It is called automatically in two situations: when *Eclipse* is executed and the plug-in is active from the previous run, or when the user activates the plug-in, either through their perspective or view. The *adts* package encapsulates the abstract concepts used in the project. It focuses on two major concepts: graphs and paths, which constitute the foundations for the whole project. The structures that use the abstract concepts from *adts* are defined in the *domain* package. It is divided in three parts: one concerning the creation of the model, other concerning coverage aspects, and a third one concerning tests. This module is the core of the plug-in; it is where the "hard work" really happens (e.g., most of the algorithms are defined in this module). In a simplified way, we can define the *ui* package as a graphical interface to the structures defined in the domain, allowing the users to provide inputs to the domain and then showing the result of these actions. This package encapsulates the decisions related to the user interface, including, the definition of graphical elements (perspective, views, and icons), as well as its behavior. Figure 4.4 gives a simplified overview of the plug-in architecture.

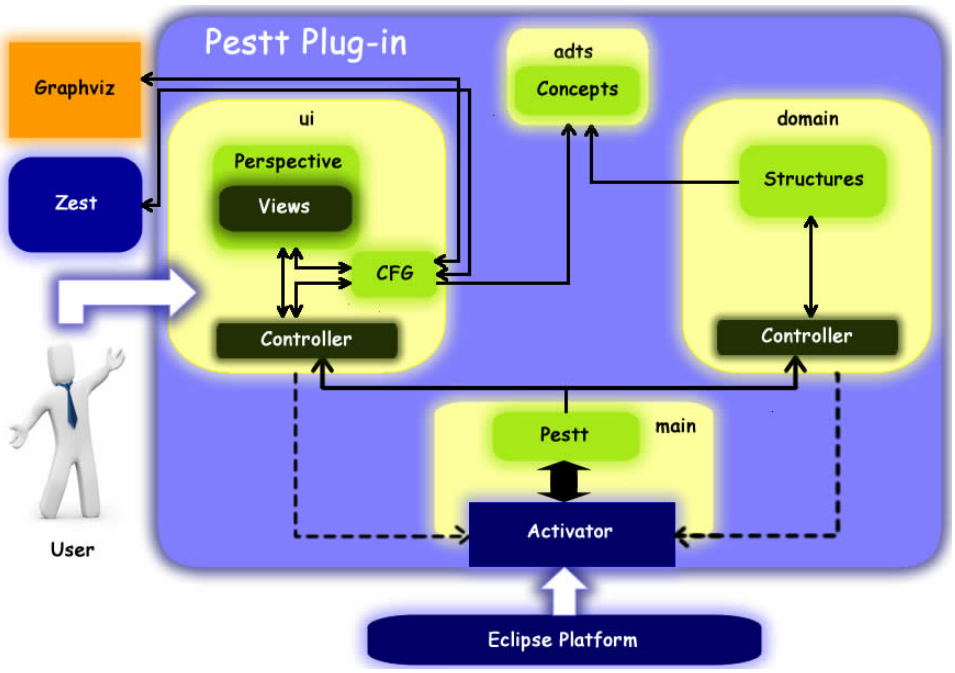


Figure 4.4: *PESTT* plug-in architecture overview

The plug-in follows the *Model-View-Controller (MVC)* pattern [28]. Internally, the communication between the various components (data representation, controllers, and views) relies in the *Observer* pattern [17, 32]. This allows us to define a one-to-many dependency between objects, so that when one object changes state, all its dependents are notified and updated automatically. Another important point is that we can encapsulate the core components in a subject abstraction, and the user interface components in an *Observer* hierarchy (allowing a separation between them). The use of these two patterns has several advantages for the development process, the most significant are:

- make it simple;
- make it faster;
- makes it more understandable;
- makes it dynamic.

Figure 4.5 depicts in more detail the plug-in architecture shown in Figure 4.4.

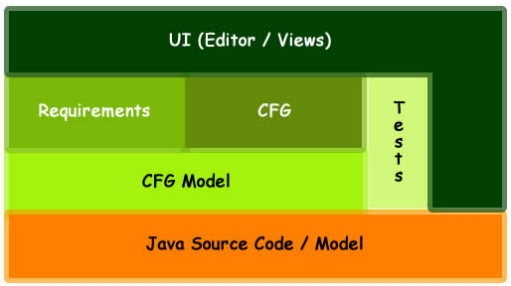


Figure 4.5: *PESTT* plug-in architecture

4.3.1 CFG: Model and Representation

The CFG model is a graph representation of the *Java* source code, which can be persisted and reviewed, but not changed. It is the main part of the whole project, so it is not surprising that most of the work has unwound around it. Besides being a representation of the *Java* source code, the CFG model is decorated with relevant information, such as the instructions of a node or guards associated with `if`, `while`, `for` and `switch` statements. As illustrated in Figure 4.5, the CFG model does not interact directly with the UI, but with three modules: CFG, requirements, and tests. The CFG representation, is the graphical representation of the model seen by the user in the UI through *Zest*. We show in detail in Section 5.1 how we build the CFG model and its representation. Figure 4.6 depicts the UML class diagram of the most important classes involved in the CFG model creation and in its representation.

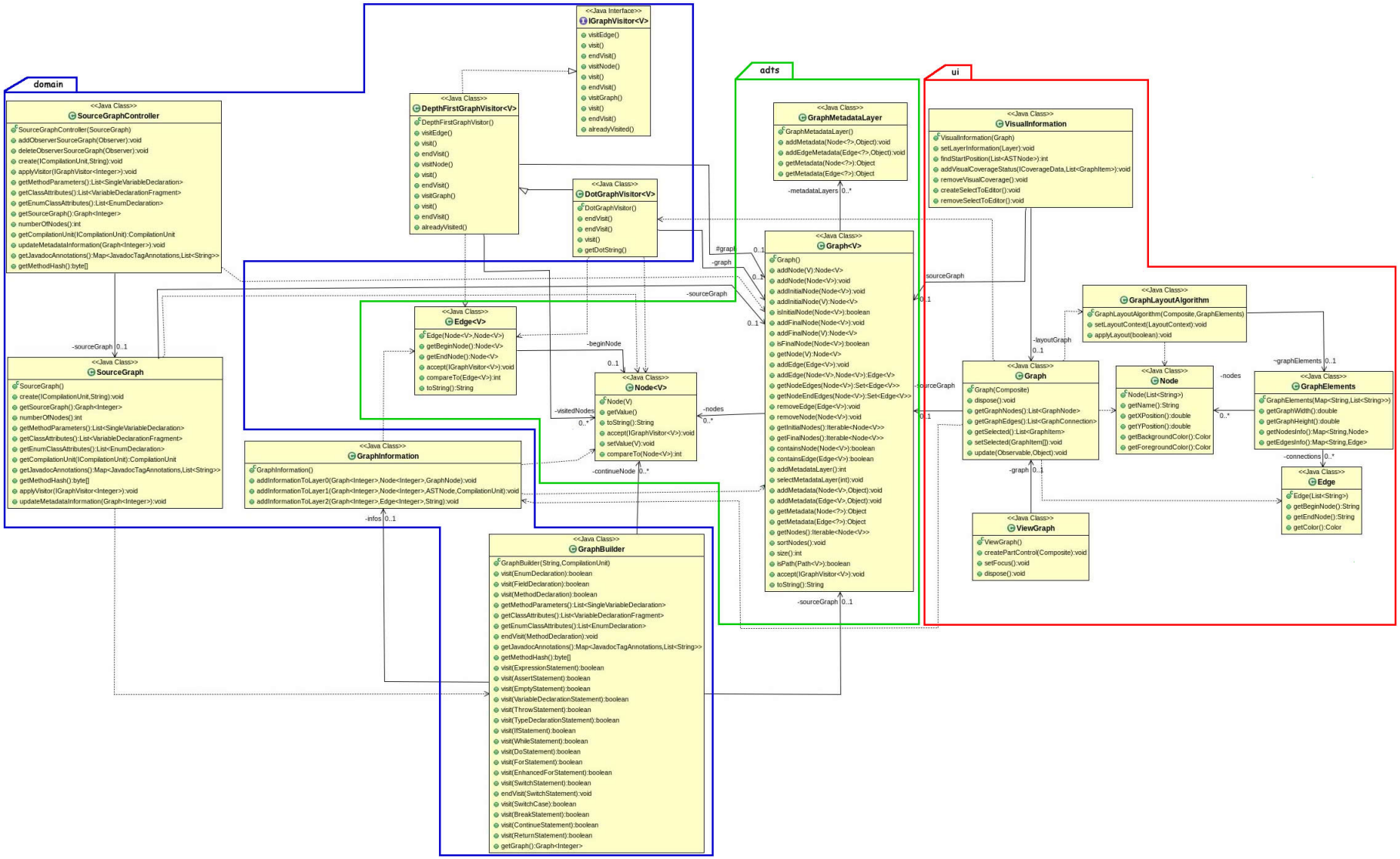


Figure 4.6: UML Class diagram for the CFG model and their representation

4.3.2 Requirements Module

Other module that interacts with the CFG model is the requirements module. It is concerned with the aspects related to the requirements set. The tasks performed by this module include the requirements set generation, and its maintenance. Internally, this module can be divided into two parts: generation, and management of the requirements set.

The requirements set is generated from various algorithms with different levels of complexity (as illustrated in Figure 2.1). However, regardless of the chosen algorithm, the behavior is similar for all: generate a set of requirements, in the form of paths from the CFG model. We follow the *Strategy* [17, 32] pattern use to design and define the algorithms generation of criteria. Then, we use the *Factory* [17, 32] pattern, by defining an interface containing the generic behavior, common to all algorithms with one object. Then, we create a set of classes that specify the particular behavior of each algorithm. Refer to Section 5.2 for implementation details of the algorithms and the relationship between this module and the CFG model. The use of this pattern is particularly suited in this situation, for several reasons:

- facilitates the integration with the plug-in—it abstracts the behavior of algorithms, keeping a common behavior interface (as if it is only one algorithm);
- makes the process more efficient, reducing the coupling, increasing code reuse (eliminating code repetition);
- makes the process dynamic—the creation details are delegated to the appropriate algorithm according to the user selection by using dynamic binding;
- makes this process more understandable and less complex;
- reduces the maintenance impact in the plug-in when algorithms are modified.

The interface with this module allows through a controller (following the *MVC* methodology). This controller maintains a reference to a set of structures that store the requirements set. These structures comprise: a set of requirements automatically generated, a set of user requirements manually managed, and a set of infeasible requirements. Automatically generated requirements are those generated by the coverage criteria algorithms. Manually managed requirements are those that have been inserted or modified by the test engineer. Infeasible requirements are those (automatically or manually generated) identified by the test engineer as being impossible to satisfy by the program execution. *PESTT* allows the test engineer to perform some operations on the requirements set, such as generate, add, remove, or edit the requirements set or mark requirements as infeasibles (see Section 4.3.4). The interaction is similar in all cases: the test engineer selects the desired operation in the GUI (e.g, add a new requirement), then the requirements controller receives the user input (for the new requirement), and forwards it to the proper class method (in this case for the manually added requirements). When processing a request two things

happen: first, the desired action is performed, and second, the requirements view is notified. Performing the desired action is self explanatory; for the case we have been describing it amounts to add the new requirement to the manually managed requirements set. Notifying the view consists in sending an object describing the changes that have occurred in the requirements set that the view needs to be update properly. Recall that the communication follows *Observer* pattern. These notifications start when one of the structures is modified and terminate when the view is updated, that is, when the changes to the requirements are displayed in the UI.

Another relevant aspect is the display of the requirements coverage status, which depends on two aspects: the test path and the selected coverage criterion.

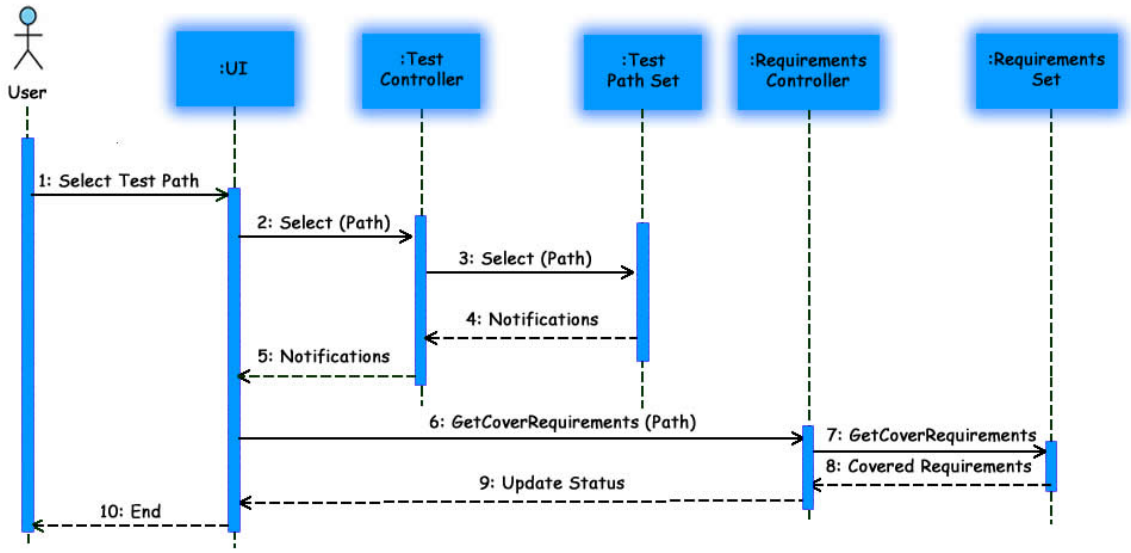


Figure 4.7: Sequence Diagram for test path selection

Here, we explain how to show the requirements coverage for a test. Section 5.2 explains the implemented algorithms to compute the requirements cover by a test path according to the selected type of cover. When the user selects a test or a set of tests (in the test view), an action is sent to the controller that dispatches it to the *domain*. This notification will be received by the requirements view (among others). After receiving the notification, the requirements view communicates with the tests controller to get the covered requirements. Internally, the tests controller communicates with the requirements controller in order to obtain the requirements covered for the selected test(s). The result is sent back to the tests controller and then to the view requirements. When the requirements view receives the set of the requirements covered, it compares them with the full set of requirements and apply the appropriate status (green if is covered, or red if is not covered) for each requirement. The final step is to show this results in the UI. The following sequence diagram shown the process described above. The UI result is:

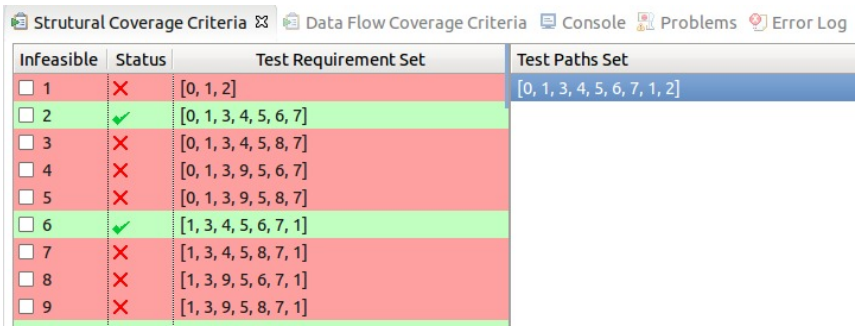


Figure 4.8: Result of the test path selection

As we mentioned earlier, the covered requirements are dependent of the selected test paths. Knowing the tests that are selected is the responsibility of the tests controller and where is the covered requirements for a test is the responsibility of the requirements controller. To communicate first with the tests controller and then with the requirements controller appears to best design decision, since it allows to maintain the concepts of test and requirement (as well as their responsibilities) separated. Similarly, we can also know which tests cover a particular requirement. Another decision taken was not to store the coverage information, because it is very volatile (it makes no sense to keep the requirements covered by a test and then choose another one). The UML class diagram shown in Figure 4.9 illustrates the package structure:

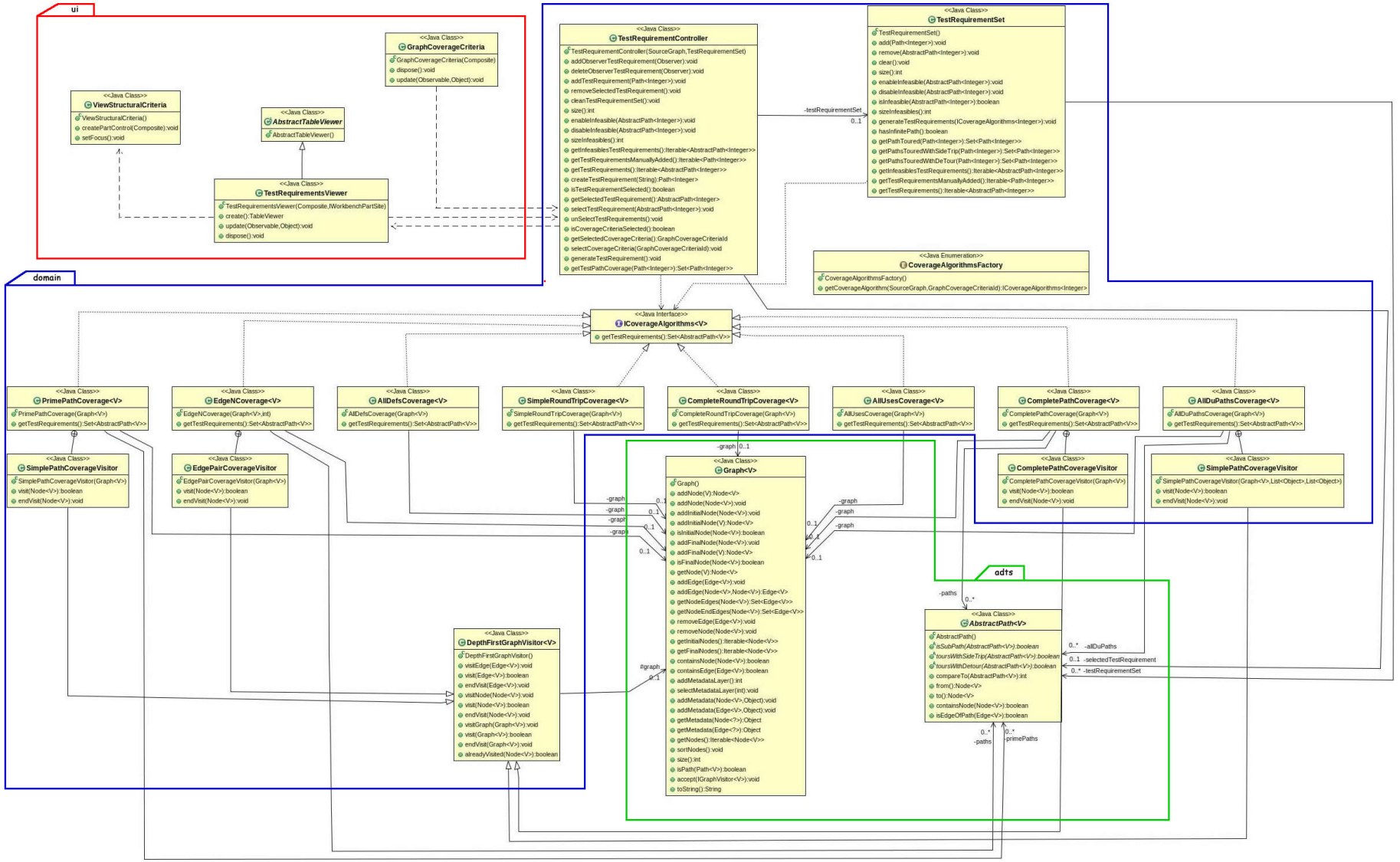


Figure 4.9: UML Class diagram for the *Requirements* module

4.3.3 Tests Module

The Test module is the responsible for the aspects concerned with test paths. This module is divided into two parts: planning and execution. First we explain the planning part. To help test engineer plan the test, was *PESTT* makes available a set of operations over test set that can be used, such as: add, edit, remove, or automatic generate test paths. Section 5.3.1 Describes the algorithm implemented to generated test paths that cover test requirements not yet satisfied. The remaining operations have a similar mode of interaction: the test engineer add, remove, or edit a path. The tests controller is the façade for these actions and notifies the corresponding views of the changes occurred in the tests set. The process ends when the view is updated, displaying the new test set.

A detail that we call the attention is for that the test engineer has full control over the test set, as well as over the requirements set. It may add, remove or edit both sets. Internally these actions are recorded including a partition between automatically generated and manually generated tests and requirements. The latter may be marked as infeasible irrespectively of being generated automatically or inserted manually. The edit operation is performed at the expense of an add and a remove operation, meaning that an edited element (test or requirement) is deleted from, and a new element is added to, the corresponding manually managed set. We chose this solution in order to maintain the structures as simple as possible.

The execution part obtains the executed paths in the CFG through test execution. This process is completely "transparent" to the test engineer: it is as simple as launching the test in *JUnit*. However, internally, it is quite complicated and involves many steps. Section 5.3.2 explains the algorithm for obtaining the executed path of a test. Here, we explain the steps to launch test execution and get the test paths. The process starts when the test engineers launches the tests using *JUnit*. The tests are launched with specific settings using the `Run As` option from *Eclipse*: called `PESTT - JUnit test`. The difference to the "`default`" launch is that we can customize the required parameters and the VM arguments. The required parameters are those present in the `default` launch, such as the project name, the test class name, the test method(s) name(s), the *JUnit* version, and the Java Runtime Environment (JRE) version. In the VM arguments, we define the command used by the *Byteman* to instrument the bytecode: `-javaagent:/tmp/byteman.jar=script:/tmp/script/rules.btm`.[2]

The command is made up one part concerning the `javaagent`, and another concerning the `script`, both indicating the location of the files within the file system, respectively, the `byteman.jar` and the `rules.btm`. The `javaagent` refers to the *Byteman* source (`byteman.jar`); the `rules.btm` specifies the set of rules defined by us, which are used by *Byteman* to instrument the bytecode (see the details in Section 5.3.2). A second file called `output.txt` is used to store the trace information

[2]Command specified for the Linux operating system.

collected during test execution. The configuration process ends with the creation of these three files; the `byteman.jar` is extracted from the *PESTT* plug-in, the `rules.btm` and the `output.txt` files are created from scratch: the `rules.btm` file contains a set of rules that are derived from the CFG. These files are created on every test launch, except the extraction of the `byteman.jar` file that is done only once. From the moment the configuration process ends, the test begins executing. In order to test execution information with each test we add a *listener* to *JUnit* execution. This listener allows us to receive notifications about the test execution process, in particular, when each test begins and when the last test ends. These notifications are received by a controller that, controls test execution and records the names of the executed tests.

The test execution proceeds as follows: the test class begins executing (let us assume that our test class has two methods *m1* and *m2*). Test *m1* begins execution, the controller receives a notification that test *m1* has started its execution. The controller stores a label "m1" in the structure which handles method name. Then continues its execution and populates the `output.txt` file with information regarding the rules (from the `rules.btm` file) that have been triggered. Then, test *m1* ends its execution and test *m2* starts. Once again, the controller receives a notification, this time informing that *m2* has started its execution. The controller stores this information as well the execution steps follow as for *m1*. Upon *m2* terminated and since there are no more tests, the test class ends the execution and the controller receives a notification that all tests have completed execution. At this point the `output.txt` file contains information regarding the execution trace of the tests *m1* and *m2*. The next step is to process `output.txt` file to infer the executed paths (Section 5.3.2 explains this procedure). For each test, the controller informs the tests controller of the test name method's and the corresponding executed test path. That stores the information and notify its observers. This process culminates with the view been updated, that is, the test paths are displayed in the UI. The tooltip of each test is the name of the test method that originate it.

Now we justify some choices made in this process. One of the options taken was to keep the name of the test method together with the executed test path. This association allows to uniquely identify in the UI, in a simple and quick way, the test method that originated the executed path. Other relevant aspect was the choice to include the `byteman.jar` file inside our plug-in and then extract it to the Operating System temporary directory. This way we reduce the plug-in configurations that are dependent on the operating system. Certainly, this might be not the ideal solution, but surely offers the best compromise between features availability and preferred settings. We decided to use a file (`output.txt`) to store the trace information collected during the test execution, because this facilitates the subsequent manipulation of the collected data. Also a test path may be arbitrarily long and are not visited to keep in it memory.

The UML class diagram shown in Figure 4.10 describes the module structure:

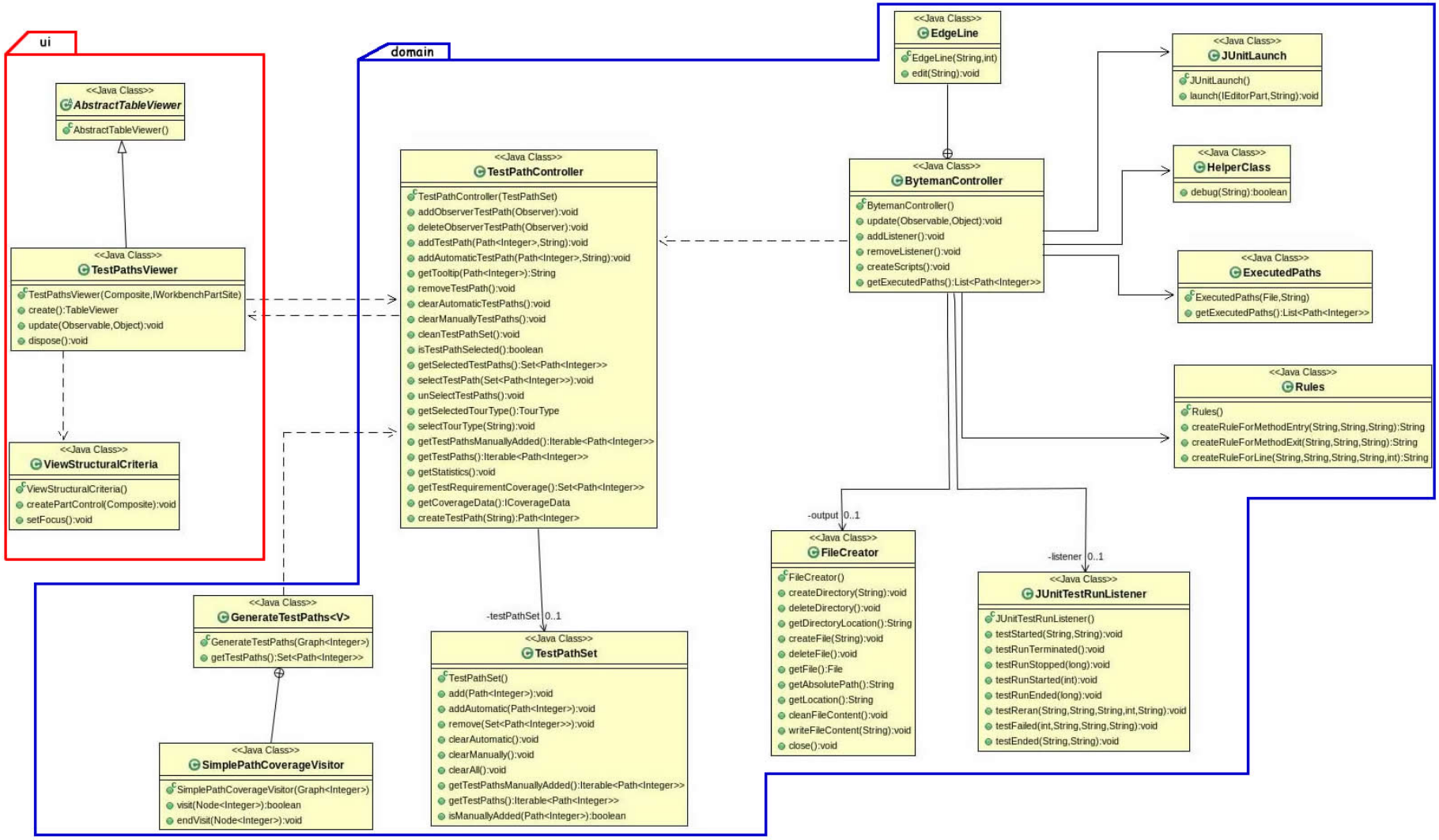


Figure 4.10: UML Class diagram for the *Tests* module

4.3.4 UI Module

Throughout the description of the previous modules we refer the graphical elements used (in a very superficial and independent way). In this module we will provide an overview of all the graphical components used and its relationship with other elements.

The plug-in consists of the following graphical elements:

- the plug-in perspective called `PESTT`.
- the `Graph` view.
- the `Graph Coverage Criteria` view.
- the `Logic Coverage Criteria` view.
- the `Structural Coverage Criteria` view.
- the `Data Flow Coverage Criteria` view.
- a set of icons in the various views.
- the *Eclipse* default views (`Console`, `Problems`, `Error Log`, `Tasks` and `JUnit`).
- the *Eclipse* editor.

The visual appearance of the plug-in is shown in Figure 4.11.

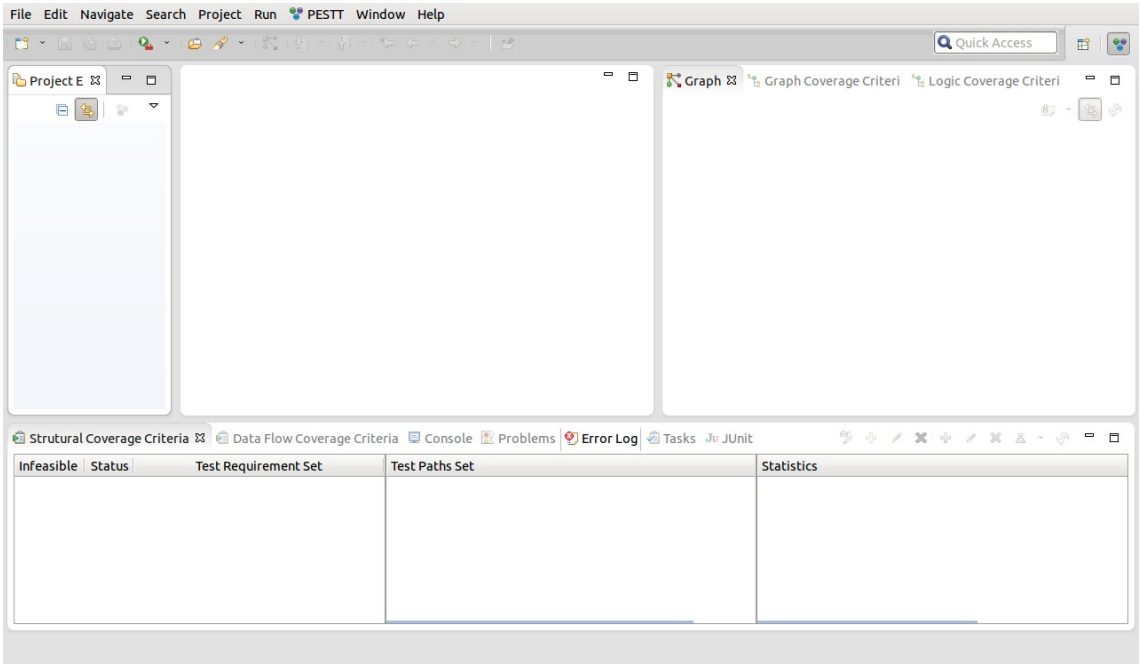


Figure 4.11: *PESTT* plug-in visual appearance

As mentioned in Section 4.1.1, all the information needed to display the plug-in in the UI (like perspectives, views and icons), is defined in the `plugin.xml` file. This is done adding a new *Extensions* to the plug-in. There are two ways of add a new extension: through the manifest UI (in the extensions page) which then saves the changes to the `plugin.xml` file or by adding the extension directly to the `plugin.xml` file. The exact list of the extensions used in the plug-in is:

```
org.eclipse.ui.preferencePages
org.eclipse.core.runtime.preferences
org.eclipse.ui.perspectives
org.eclipse.ui.views
org.eclipse.ui.perspectiveExtensions
org.eclipse.ui.commands
org.eclipse.ui.menus
org.eclipse.ui.bindings
org.eclipse.ui.handlers
org.eclipse.core.resources.markers
org.eclipse.ui.editors.annotationTypes
org.eclipse.ui.editors.markerAnnotationSpecification
org.eclipse.debug.ui.launchShortcuts
```

Listing 4.2: *PESTT* list of extensions.

We present part of `plugin.xml` file in Appendix A (Listing A.3). These extensions allow us to define the elements that we use, specifying the name, icon, and its position in the perspective. However, the definition of the extensions is only part of the process, the other part is to define the actions that each element must execute.

PESTT Perspective

We choose to create a perspective in order to be able to concentrate the different views used, this allows us to load automatically all the views with a single operation. The perspective can be opened through ($Window \rightarrow Open\ Perspective \rightarrow Other... \rightarrow PESTT$). To integrate the *Eclipse* default views in the perspective we need to add the following code, this allows us to define the exact position of each view in perspective:

```
String editorarea = IPageLayout.ID_EDITOR_AREA;
IFolderLayout bottom = layout.createFolder("bottom", IPageLayout
    .BOTTOM, 0.75f, editorarea);
bottom.addView(Description.VIEW_STRUCTURAL_COVERAGE);
bottom.addView(Description.VIEW_DATA_FLOW_COVERAGE);
bottom.addView(IConsoleConstants.ID_CONSOLE_VIEW);
bottom.addView(IPageLayout.ID_PROBLEM_VIEW);
bottom.addView("org.eclipse.pde.runtime.LogView");
bottom.addView(IPageLayout.ID_TASK_LIST);
bottom.addView("org.eclipse.jdt.junit.ResultView");
IFolderLayout left = layout.createFolder("left", IPageLayout.
    LEFT, 0.15f, editorarea);
left.addView(IPageLayout.ID_PROJECT_EXPLORER);
```

Listing 4.3: *PESTT* list of extensions.

The remaining views are integrated into perspective through its definition in `plugin.xml` (see in Appendix A (Listing A.3) line 41-57 how we did to the `Graph` view). The different views can be loaded independently of the other views through ($Window \rightarrow Show$

$View \rightarrow Other... \rightarrow PESTT$) and then choose the desired view.

Graph View

The `Graph` view is the main view of the plug-in, it is used to show the CFG. Internally it is composed of only one element (the CFG), it offers three actions (available via the icons): draw the CFG, link the graph with the *Eclipse* editor and show extra information in the CFG.

Draw the CFG will be explain in Section 5.1, but as the name suggests allow the users to draw and visualize the CFG corresponding to *Java* source code.

Link the graph with the *Eclipse* editor, allow the users to select an element in the CFG (node or edge) and the corresponding *Java* source code associated with it become highlighted, the opposite also occurs, selecting a piece of code the corresponding element in the CFG (node or edge) associated with it will be highlighted as well. This was done using two listeners (one in *Eclipse* editor and another in the CFG). These listeners are active only when the option is active, and allow us to detect precisely the element(s) that the user select. This information is then passed to a controller that perform the necessary procedures to highlight the element (in the CFG or in the code according to the user's choice).

The third option allow the users to add extra information to the CFG, more specifically, guards, breaks, etc. In order to facilitate the understanding of CFG when the extra information is visible, the user can control the information that wants to view by choosing one of the three options available: *Empty* (don't show any extra information), *Guards (All)* (shows all the extra information available) or *Guards (Custom)* (allow the user to customize the information that will be show, this can be done in two different ways: *True branches* (shows only the true branches) or *False branches* (shows only the false branches)). Selecting an option, the controller is notified, perform the necessary procedures. Changes are made and a notification is sent back to the view indicating to it to update.

This view (more precisely the CFG) is linked to all other views in the plug-in (those created by us and the *Eclipse* editor). The `Graph` view can be seen in the upper right of the Figure 4.11.

Graph Coverage Criteria View

The `Graph Coverage Criteria` view includes an exact replica of Figure 2.1. This view has two distinct functions: one is to let the user choose the coverage criterion to apply and the other is to make users aware of the relationship between the various criteria. This view internally use *Zest* to represent the criterias, the Listing A.4 illustrates how this representation has been made.

Logic Coverage Criteria View

This view is equal to the `Graph Coverage Criteria` but applied to the coverage criteria based on Logic expressions. This view was used for educational purposes only (the implementation of the algorithms shown in this view, is not part of the plan for this thesis, but all base structures that support the algorithms have already been implemented).

Structural Coverage Criteria View

This view is the most complex of all, it is used to show the requirements set, the test paths set and the coverage statistics, all representations used in this view are in accordance with the concepts introduced in the Chapter 2.

Inside, this view is divided into three separate parts (one for each of the components listed above) as can be seen at the bottom of the Figure 4.11. Respectively, on the left side the requirements view, the tests view at the middle and the statistics view on right side. All views operate independently of each other and all work the same way internally. The structure of all views is based on a table. The requirements view consists of three columns, respectively one to indicate whether the path is infeasible, other to indicate the requirement coverage status and another to show the requirements. Choose if a particular requirement is infeasible is as simple as marking a check-box. The requirement coverage status are shown through images. The requirements are presented as text. Both the tests view and the statistics view are composed of only a single column of simple text.

The requirements view and the tests view are linked to each other and to the `Graph` and `Data Flow Coverage Criteria` views (when applicable), we did this by using the *Observer* pattern. We can illustrate this with a simple example. Both test and requirements are paths created from the CFG (as we shall see in Chapter 5), when a test is selected (in the test view), in addition to the process explained in the Section 4.3.2 (which causes the requirements to become painted of green if they have been covered or red if they have been not covered), the same notification is also received by the CFG controller which perform the necessary procedures to highlight the path in CFG (the same happens if a requirement is selected). Another view that is notified is the statistics view which shows textual coverage statistics for the selected test (the statistics view is a purely informative view that does not allow any type of interaction with the user or with other views, it only presents any kind of information if we select a test, it does not works if we select a requirement).

We chose this structure for the views because it allows us to integrate all the elements in a single place and it allows the user to interact with the program in a a simple and intuitive way. The use of tables allows us a better view of the elements and enabling other structures within it (such as the check-boxes used to mark requirements as infeasibles). Another reason which has led us the use of tables was the fact that we can define the type of selection, vary the range of selection (one to many) in the tests view, allow the user

not only have a vision of the requirements covered by only one test but also gives and overview of all requirements covered by a group of tests.

Finally, we need to mention that the interaction between the users and the sets (requirements and tests, in other words, with the requirements and tests modules) is done through the icons present in Figure 4.11 (on top of statistics view). The icons perform the following actions:

- generates the test paths set that covered all requirements.
- add a new requirement path.
- edit a requirement path.
- remove a requirement path.
- add a new test path.
- edit a test path.
- remove a test path.
- select the tour type (one of the following: *Tour*, *Tour with Sidetrips* or *Tour with Detours*)
- generates the requirements path set based on the chosen graph coverage criteria.

Most of the features already were described in the Sections 4.3.2 and 4.3.3, those not yet been presented will be in the Chapter 5.

Data Flow Coverage Criteria View

This view is used to show the flows of data values, displaying to the user in a simple and understandable way the variables definitions and their uses (as introduced in Section 2.2.2.2).

Internally, the results can be presented in two different ways: one focused on the variable and other focused on the CFG elements (node or edge). In the variable view, for all variables are showed the CFG elements (node or edge) where the variable was defined and used. In the CFG elements view, for each CFG element are showed the variables that are defined and those that are used.

The structure of this views is based on a table. The views are composed of three columns of simple text (in one case we have Variables, Definitions and Uses, in the other we have Node/Edge, Definitions and Uses). The choice of how the results appear (focused on the variable or focused on the CFG elements) is fully controlled by the user (through an icon), the users can change the display mode at any time without the need to generate the values all over again.

This views are linked to the `Graph` view and `Requirements` view (only when the criterion selected is one of the *Data Flow Coverage Criteria*). In the same way as explained previously, when one of the elements is selected in either views the corresponding elements in the CFG become highlighted.

Finally, we need to mention that the interaction between the users and the program is done through the icons. The icons perform the following actions:

- choosing the view (one of the following:*Variable* or *Node/Edge*)
- obtain the program data flow.

In the Section 5.2.2 will be explained in detail how the data flow was obtained.

Eclipse editor

Although not a view developed by us, the *Eclipse* editor view, plays an important role in our plug-in. It is through it we saving (in the form of *Javadoc* comments) all actions that users carry out, such as: add, edit a requirement or a test path, the tour type currently in use, the selected criteria as well as the infeasible requirements. Using this feature allows us to recover the deeds done between sessions. This process like all the others also works with *Observer* pattern, this means that whenever the user makes a change, the editor is notified to update the *Javadoc* with the new options.

The UML class diagram shown in Figure 4.12, gives an idea about the module structure:

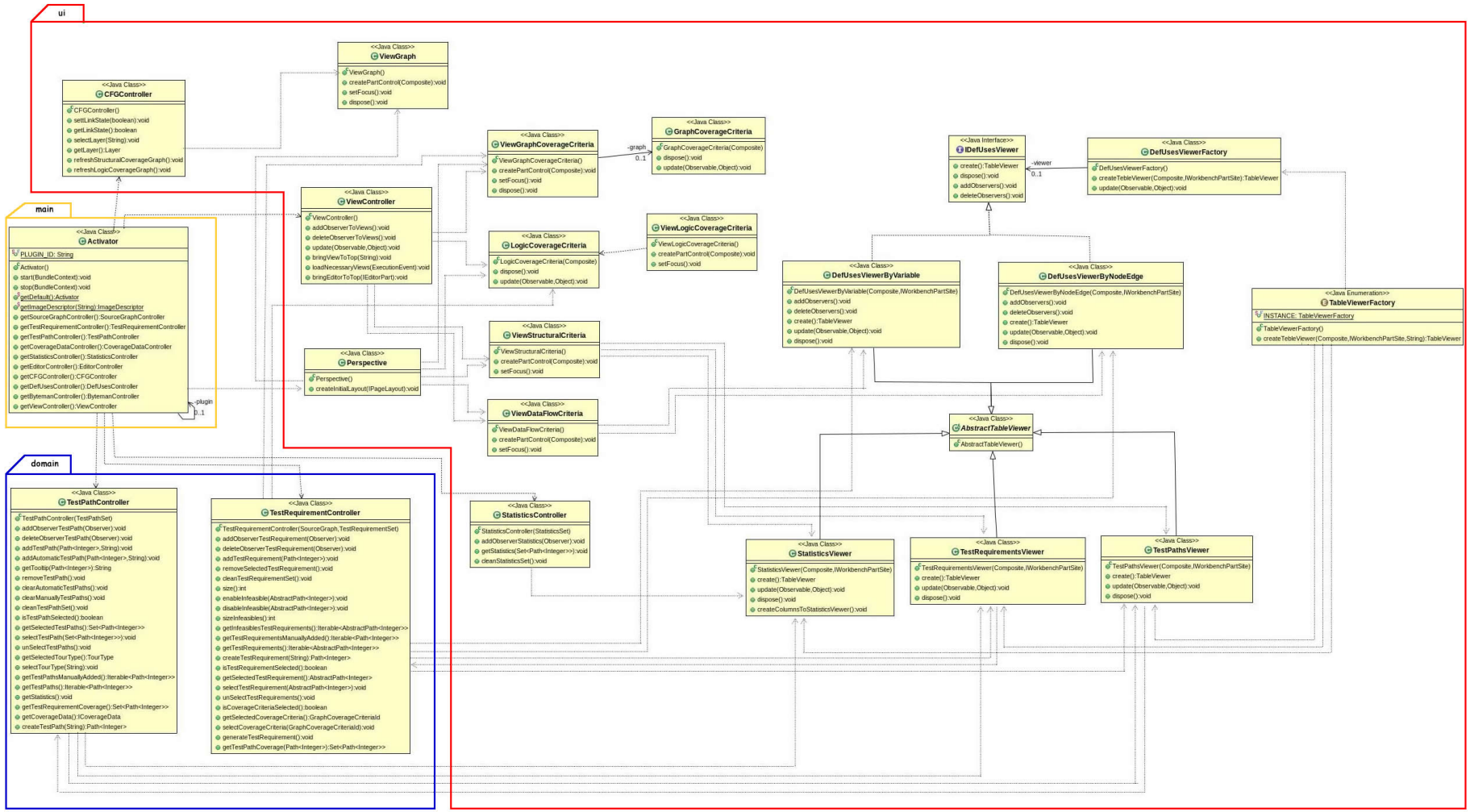


Figure 4.12: UML Class diagram for the *UI* module

Chapter 5

Algorithms

In this chapter we present the most important algorithms implemented during the course of this thesis.

5.1 CFG Builder

The way to build the CFG takes the real effort. It is build on *Java* classes that contain methods, attributes, and other *Java* elements. If we analyze the *Java* source code we can clearly see its nested structure. For example, if an element is followed by a bracket block then, most probably, it is a parent element with descendants; otherwise its a simple construct. As long as the elements do not have a special meaning, the data structure of the model is clear. Things start to get a little more complicated, leading to more complex CFG, when structures containing ”jumps” to other sibling, children, parent, or even other far aways constructs. This is the case for loops contain `break` or `continue` statements or *Switch-Cases*, where the `break` statements influence significantly the next statement to be executed. For now we know that our model is a tree, (an Abstract syntax tree) that needs to be mapped into a CFG.

5.1.1 The CFG Model

To describe the construction of the CFG model, we first introduce the elements used. The CFG model is based on three concepts:

- Node;
- Edge; and
- Graph.

Nodes are the graph basic units. Associated with it we have some features, where we highlight the following two: set and get the node value. As such, nodes are mutable structures where its value serves as the node identifier. Also, nodes are comparable and visitable. CFGs, we need to know the direction of the flow).Edges capture this notion

by having a source and a destination node. Other features associated with edges are to get the source node and to get the destination node. Similarly to nodes, edges are also comparable and visitable. Internally, a graph object consists of a list of nodes and a map of edges, associating a node its successors. Of all Graph features, the most important are: add/remove a node, add/remove a edge, add/remove an initial/final node, set/get the initial/final nodes, get a list of all nodes, get a set of all edges, get a set of edges that start in a given node, get a set of edges that end in a given node, add extra information to graph/nodes/edges.

These three concepts have been defined in a generic way, in order to support nodes with values of any data type (although we are only interested in *Integer* value nodes). This makes the model completely independent from the data type used (which does not imply modifications in the model if the data type for node value changes). Moreover, it allows the model to be reused in other situations. The three implementations only define the generic behavior expected for these concepts, however, certain details are not the responsibility of the model (e.g., prevent nodes from having the same value or prevent edges from connecting the same nodes). This kind of situations are the responsibility of the structures that implement the model.

Now its time to present how the CFG is built from the *AST* of the *Java* method.

The process:

- gets the *Java* source code;
- parses the *Java* source code into the *Abstract Syntax Tree (AST)*; and
- traverses the *AST* and builds the CFG model.

Eclipse offers a toolkit called *Java Development Toolkit (JDT)* to handle operations with *Java* source code. Internally, *JDT* has a compiler called *Eclipse Compiler for Java (ECJ)* that during the compilation process builds an abstract representation of the *Java* source code—an *AST*. The *AST* maps the *Java* source code into a tree, identifying the elements present in the *Java* class (their names and attributes) hierarchically. This *AST* can be viewed using the *AST View* plug-in for *Eclipse*.

To have access to the *AST*, the process of parsing the *Java* source code must be controlled by us. This process is automatic, we just need to know the file corresponding to the *Java* class for which we want to obtain the *AST*. Listening 5.1.1 shows how we obtained the *AST*.

```
1 IEditorPart part = PlatformUI.getWorkbench().
     getActiveWorkbenchWindow().getActivePage().getActiveEditor();
2 IFile file = (IFile) part.getEditorInput().getAdapter(IFile.
     class);
3 ICompilationUnit compilationUnit = JavaCore.
     createCompilationUnitFrom(file);
4 ASTParser parser = ASTParser.newParser(AST.JLS3);
5 parser.setKind(ASTParser.K_COMPILATION_UNIT);
```

```
6 parser.setSource(compilationUnit);
7 parser.setResolveBindings(true);
8 CompilationUnit unit = (CompilationUnit) parser.createAST(null);
```

Listing 5.1: Parsing the *Java* source code into the *AST*.

With the *AST* built, the best way to traverse the tree is by using the *Visitor* pattern [17, 32], more specifically an *ASTVisitor*. The *ASTVisitor* is an abstract class that implements a general depth first visitor for the *Java AST*. This visitor is based on the template method[17, 32] pattern and offers two methods: `visit` and `endVisit`. It allows for pre order and pos order traversing, and whenever the visit method returns false, its subtree is not visited.Our implementation of the *AST* visit for building CFG is called `GraphBuilder`.

Although we have access to all *AST* nodes, we are only interested in the part corresponding to the method selected by the user. The visitor traverses the *AST* in a depth first way. It all and adds the defined elements to the graph model. Now, we detail the visitor for the interesting *Java* elements:

visit(MethodDeclaration node)

When a method declaration is encountered it (the visitor) checks whether the method is the one selected by the test engineer. If so, the visitor traverses the method body; otherwise, it skips to the next method declaration.

endVisit(MethodDeclaration node)

When the method declaration ends the visit, the CFG model is optimized (see Section 5.1.2) and is ready to be shown (see Section 5.1.3).

visit(ExpressionStatement node), visit(AssertStatement node), visit(EmptyStatement node), visit(VariableDeclarationStatement node), visit(TypeDeclarationStatement node)

When a expression-statement, or an assert-statement, or an empty-statement, or a variable-declaration-statement, or a type-declaration-statement is encountered the information associated with it is added to the current node.

visit(BreakStatement node)

When a break-statement is encountered a new node is inserted into the model. The new node is connected to the predecessor node (implies that there is at least one node in the model)—the predecessor node is the last node inserted to the CFG, e.g., the predecessor node is the node corresponding to if-statement:

```
for
  if
    break;
```

(The first node created, in the below descriptions, is connected to the predecessor node.)

Break-statement is added to the node. There is no difference between the loop-break and the switch-break. Cases are treated differently by the time the graph is build. Contains no child nodes.

visit(ContinueStatement node)

When a continue-statement is encountered a new node is inserted into the model. The new node is connected to the predecessor node . Continue-statement is added to the node. Contains no child nodes.

visit(ThrowStatement node)

When a throw-statement is encountered a new node is inserted into the CFG. Throw-statement is added to the node. Contains no child nodes. This is a final node.

visit(ReturnStatement node)

When a return-statement is encountered a new final node is inserted into the model. Return-statement is added to the node.

visit(IfStatement node)

When a if-statement is encountered a node is inserted to the if-statement, another to the then-statement, another to the else-statement whether an else branch exists, and another to mark the end of the if-statement. The then-statement and the else-statement may contain child nodes and are connected to the if-statement node. The node that indicates the end of the if-statement is connected to the then-statement node and to the else-statement node (when it exists; otherwise it is connected directly to the if-statement node). Information about each statement is added to the corresponding node; information about guards is also added to the edges that connect the if-statement to the `then` and the **`else`** branches.

visit(WhileStatement node)

When a while-statement is encountered one node is inserted to represent the while-statement, other to represent the while-body, and another to indicate the end of the while-statement. The node corresponding to the while-statement is connected to the node that

indicates the end of the while-statement. The while-body is connected to the while-statement node and may contain child nodes; information about guards is also added to the edges that connect the while-statement to the other nodes.

visit(DoStatement node)

When a do-statement is encountered one node is inserted to represent the do-body, other to represent the while-statement, and another to indicate the end of the while-statement. The node corresponding to the while-statement. The while-statement is connected to the node that indicates the end of the while-statement; information about guards is also added to the edges that connect the while-statement to the other nodes.

visit(ForStatement node), and visit(EnhancedForStatement node)

When a for-statement is encountered one node is inserted to represent the for-statement, other to represent the for-body, other to represent the for-increment, and another to indicate the end of the for-statement. The for-statement variables initializations are added to the predecessor node. The for-statement node is connected to the node that indicates the end of the for-statement. The for-body is connected to the for-statement node and to the for-increment node; the for-increment node is also connected to the for-statement. Information about guards is also added to the edges that connect the for-statement to the other nodes.

When a for each loop is encountered one node is added to represent the enhanced-for-statement, other to represent the enhanced-for-body, and another to indicate the end of the enhanced-for-statement. The enhanced-for-statement is connected to the node that indicates the end of the enhanced-for-statement. The enhanced-for-body is connected to the enhanced-for-statement node. Information about guards is also added to the edges that connect the enhanced-for-statement to the other nodes.

visit(SwitchStatement node)

When a switch-statement is encountered a node is inserted the represent switch-statement and another to indicate the end of the switch-statement.

visit(SwitchCase node)

The switch-case-statement represents the switch-statement-body—the `switch case` or the `switch default`.When a switch-case-statement is encountered a new node is inserted into the model for both the `case` and the `default`. The new node is connected to the switch-statement node. Information about the switch-case-statement is added to the node; information about *case/default* is also added to the edges that connect the switch-statement to the other nodes.

endVisit(SwitchStatement node)

When a switch-statement ends the visit, the node that indicates the end of the switch-statement is connected to all the nodes inserted into the model in *visit(SwitchCase node)*.

The `GraphBuilder` implements the abstract class `org.eclipse.jdt.core.dom.ASTVisitor` and builds the CFG model.

The generated graph has only a single initial node, reflecting the methods single entry point and multiple final nodes. Some cases that lead to the creation of final nodes are: the existence of a return-statement, the existence of a throw-statement, or the syntactic method end. Considering these facts, we can say that the CFG has at least one node. That may initial and final at the same time.

Next, we illustrate the algorithm through an example using the Listing 3.1 (Section 3.2.3). A portion of the *AST* for this method is shown in Listing A.5 (Appendix A). Because of the nature of the *Java* "constructs" algorithm maintain predecessor nodes stored in a stack (we call it `prev- Node`). This process allows us to built the CFG in a compositional way, facilitating the nodes management and, in any situation, we have the perception of the existing dependencies. The model starts with *node 0*, this node is added to the `prevNode` stack. Going down in the *AST* we found the method declaration (`visit(MethodDeclaration node)`); since the method is the one selected (the example has only one method) we visit its body. The first instruction in the method is a for-statement, so `visit(ForState- ment node)` is executed. This method adds the variables initializations to the node at the top of the stack (*node 0*). Then, *node 1* is created—corresponding to the for-statement—whenever a new node is created its value is incremented one unit relative to the value of the previous node. After that, a connection between the *node 0* and *node 1* is created. Creating a connection (an edge) involves two operations: the first is the removing of the node at the top of the `prevNode` and the second is the inserting of the edge destination's node into the stack, in other words, *node 0* is popped and *node 1* is pushed into, the stack.

The next step is to add the for-statement information to the *node 1*. This process consists in store the instructions associated with the for-statement in *node 1*. This data is stored in a graph layer called `INSTRUCTIONS`, which keeps the instructions associated with each node.

Java allows `breaks` or `continues` instruction to occur in loop-statements. In order to support these instructions we use two extra stacks: one for storing nodes for the break-statement (`breakNode`) and the other for continue-statement nodes(`continueNode`)). These two stacks allow us to pair the break/continue with the closest loop in a compositional way. If a break-statement occurs the node created by it must be connected to the node that indicates the end of the loop; if a continue-statement occurs the node created by it must be connect to the body node of the loop. Getting back to the example, the

for-increment node is inserted into the model (*node 2*) and the variable increment is associated to it (x++ is added to *node 2* `INSTRUCTION` layer). Then, the node that indicates the end of the for-statement is inserted into the model (*node 3*). Usually, the node that indicates the end of a statement has no information associated. At this point we do not know what comes next, so this nodes has to be preserved, this means that the *node 2* is pushed to the `continueNode` and *node 3* is pushed to the `breakNode`. Add *node 3* to the `breakNode` stack is no surprise, since when a break-statement is executed inside a loop it automatically ends and the program continues its execution at the first line after the end of the loop. *Node 2* is added to the `continueNode` if a continue-statement is executed, the next instruction executed is the increment of the variable. The model continues from the for-statement node, in other words, *node 1* is added to the `prevNode`.

When the for-statement guard is evaluated to *true*, the for-body is executed. A connection between the for-statement and for-body is created (an edge between *node 1* and *node 4*). This means that *node 1* is poped from `prevNode` and *node 4* is pushed into the stack. Information is also added to some edges, more specifically, the edges that represent branching. This data is stored in a graph layer called `GUARDS`. So, $x <= 3$ is added to the edge that connects *node 1* to *node 4*. *Node 4* is added to the `prevNode`.

Then, we continue visiting the *AST*, to the for-body. At this point, the execution of the method `visit(ForStatement node)` is suspended. The for-body has two statements, more specifically, two if-statement, so the method `visit(IfStatement node)` is executed. The first thing this method does is to create a connection between the node in the top of `prevNode` and the if-statement node (an edge between *node 4* and *node 5*). This means that *node 4* is removed from the `prevNode` and *node 5* is inserted into the model. The information about the if-statement is added to the *node 5*, this data are stored in the graph `INSTRUCTIONS` layer. *Node 5* is added to `prevNode`. The if-statement has always a then-statement (when the guard is evaluated to *true*), in practice this results in a connection between *node 5* and *node 6*. Since this edge represents a decision, $x \ \% \ 2 == 0$ is added to it. This data are stored in the graph `GUARDS` layer. *Node 6* is added to `prevNode`.

We continue going down in the *AST*, in order to visit the then-statement, we do this through the following instruction `node.getThenStatement().accept(this)`. At this point the method `visit(IfStatement node)` is suspended. In the then-statement, we have only an expression-statement, this means that the next method to be executed is `visit(ExpressionStatement node)`. This method only adds the expression to the node at the top of the `prevNode`, in practice this corresponds to add *System.out.println("pair")* to *node 6*. The `visit(ExpressionStatement node)` ends and because it is a leaf in the *AST* we cannot continue going down, so the execution returns to the `visit(IfStatement node)` method (remember that the visitor traverses the *AST* in a depth first way).

The value of the occurrence of interruptions (break-statement, continue-statement, throw-statement and return-statement) are saved, in order to make the correct connections in the model. After this, *Node 5* is added to `prevNode`, so we can define the other connection of the if-statement (when the guard is evaluated to *false*). In the `prevNode` are stored the *node 6* and the *node 5*. Here we have two possibilities: the if-statement has an else-statement or not. If it has (such as this case) is created a connection between the node in the top of `prevNode` and the else-statement node (in the model corresponds to an edge between *node 5* and *node 7*), this means that *node 5* is removed from the `prevNode` and *node 7* is inserted into the model. Since this edge represents a decision, $\neg(x \ \% \ 2 == 0)$ is added to it. This data are stored in the graph `GUARDS` layer. *Node 7* is added to `prevNode`.

The visit process of the else-statement is equal to the then-statement, but is done through the following instruction `node.getElseStatement().accept(this)`. Which means, that *System.out.println("odd")* is added to *node 7*. Like in the then-statement the value of other interruptions (break-statement, continue-statement, throw-statement and return-statement) are saved too.

The final step of the if-statement is to determine how it ends. The way the if-statement ends depends on the occurrence of interruptions in the then-statement and in the else-statement (when it exists). If the values stored for both (for the then-statement and for the else-statement) contains a return-statement, then the if-statement is complete. Otherwise, the node that indicates the end of the if-statement is created (a connection between the node at the top of the `prevNode` and the node that indicates the end of the if-statement is created, this connection can be between the if-statement (or the else-statement when it exists) and the node that indicates the end of the if-statement. If the connection is between the if-statement and the node that indicates the end of the if-statement (an edge that represents a decision) the information about the guard is added to it, and then is stored in the graph `GUARDS` layer.

Then the values are verify again, but this time to check the occurrence of breaks/continues. If breaks/continues are not found, then a connection between the node at the top of the `prevNode` and the node that indicates the end of the if-statement is created, this represents a connection between the then-statement and the node that indicates the end of the if-statement, otherwise the connection was already created by the appropriate method during the visit.

Returning to the example, at the `prevNode` are stored *node 6* and *node 7* corresponding respectively to the then-statement and the else-statement. Since interrupts are not detected, the node that indicates the end of the if statement is inserted into the model (*node 8*). Connections between *node 6* and *node 8* and between *node 7* and *node 8* are created. After the connections have been created, the `prevNode` is empty. Note that despite being created two connections that end at the *node 8*, it is inserted into the model

only once. *Node 8* is added to `prevNode`, in order the model proceed from here.

The first if-statement ends its execution, and the visitor proceed traverses the *AST*. Then the second if-statement is reached. The above process is repeated. Four new nodes are added to the model (such as edges and their corresponding information). When the if-statement ends it execution, `prevNode` contains the *node 12*.

Since the for-body has no more children we cannot continue going down in the *AST*. So the execution returns to the `visit(ForStatement node)` method. When the for-body ends its visit, the `continueNode` and `breakNode` are cleaned (because the connections have been created during the visit by the appropriate methods). Then similar to that mentioned in the if-statement, is checked if interruptions occurred so the appropriate connections can be defined. If there were no interruptions, a connection between the node at the top of the `prevNode` and the for-increment node is created, otherwise nothing is done. Thereafter, the remaining connections are created, more specifically, between the for-increment node and the for-statement node and between the for-statement node and the node that indicates the end of the for-statement. This last connection represents a decision (when the guard is evaluated to *false*), this means that the information about the guard is added to it, and then is stored in the graph `GUARDS` layer.

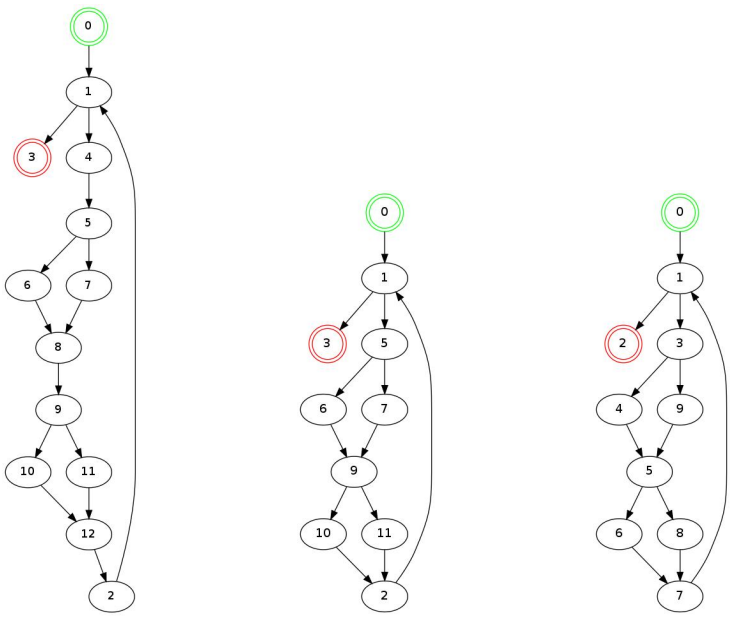
In the example, this corresponds to the following three connections: one between *node 12* and *node 2* (corresponding to the connection between the node in the top of the `prevNode` and the for-increment node), other between *node 2* and *node 1* (corresponding to the connection between the for-increment node and for-statement node) and another between *node 1* and *node 3* (corresponding to the connection between the for-statement node and the node that indicates the end of the for-statement), to the last connection is added $x <= 3$. With this step the model is nearly complete, and its representation is shown in Figure 5.1a. It only remains to optimize the model, a process that is described in the following section.

5.1.2 CFG Model (Optimization)

The model optimization process is divided into two distinct stages. In the first stage are removed all the elements (nodes and edges) that are unnecessary to the model. An element is considered unnecessary if it is an empty node (except if it is a final node), this means that the node has no information associated to it in the graph `INSTRUCTIONS` layer. Consequently, are also considered unnecessary elements all the edges that are associated with the empty nodes (are they incoming or outgoing edges).

We start by identify the set of empty nodes present in the model, by checking if they don't have information associated with it in the graph `INSTRUCTIONS` layer. Then we identify for each node the edges associated with it. The next step is create a new connection between the source node of the incoming edges and the destination node of the outgoing edges (where the node is the destination node of the incoming edges and where

it is the source node of the outgoing edges). After this procedure have been performed by all the identified nodes, the nodes are removed from the model (as well as the respective edges). Let's look to the example. The empty nodes present in the model are: *node 4*, *node 8* and *node 12*. The first empty node is *node 4*, which is associated with the following edges *(1, 4)* and *(4, 5)*, a new connection between *node 1* and *node 5* is created. Since *node 4* has no more edges, the process advances to the next node on the list. *Node 8* is associated with the following edges *(6, 8)*, *(7, 8)* and *(8, 9)*, in this case two new connections are created: one between *node 6* and *node 9*, and another between *node 7* and *node 9*. The process advances to the next node, *node 12* is associated with the following edges *(10, 12)*, *(11, 12)* and *(12, 2)*, two new connections are created: one between *node 10* and *node 2*, and another between *node 11* and *node 2*. Now all empty nodes (and the respective edges) are removed from the model. At this point we have an optimized model, but which does not have a set of sequential nodes (after the empty nodes have been removed, the model have the following nodes: *node 0*, *node 1*, *node 2*, *node 3*, *node 5*, *node 6*, *node 7*, *node 9*, *node 10*, *node 11*) (see Figure 5.1b), this is where the second stage of the optimization process starts. At this stage the model is rewritten so that we can have a set of sequential nodes. This process consists in traverse the model and rewrite the values of the nodes in a sequential way. To do this we create a visitor that traverses the model in a depth first way, and as it goes down in the model, the values for the nodes will be updated. The optimization process occurs at the end of the method visit, in other words in the method `endVisit(MethodDeclaration node)`.



(a) CFG model (non-optimized).

(b) CFG model (optimized).

(c) CFG model (final).

Figure 5.1: The different representations of the CFG model during the build process

In the model, nodes are only concepts that have no graphical representation, its distinction is made through specific structures that allow us to identify the type of node. Note that were used different graphical representations to the nodes, in order to distinguish the initial and final nodes from the other nodes (only for better understanding of the model).

5.1.3 CFG Representation using *Zest*

Now that the model is built, the next step is display it to the users. To do this, is necessary create a representation of the model, a CFG. As mentioned in previous chapters we use *Zest* to view the CFG, which means, that we use the API provided by *Zest* to create the CFG, more specifically we use the `Graph(parent, SWT.NONE)` (this creates an empty graph). In order to add the elements of the model to the CFG was created a visitor. This visitor traverses the model in a depth first way, and as it moves through the model it collects information. This information represents the graph specification (similar to that shown in Figure 3.1a). The specification contains the model edges and the nodes appearance. The CFG can be build only with the edges, because it have the information about the nodes and their connections, however due to the limitations relative to the appearance, we had to include extra information about the nodes appearance. This extra information is purely stylistic and allows us to define the color of the nodes (since it is not allowed in *Zest* define the shape of the nodes, and we need a way to distinguish the initial and the final nodes from the others). Nevertheless this is not sufficient to build a noticeable CFG (because of the algorithms provided by *Zest*). The lack of an algorithm able to present a noticeable CFG, led us to choose to use Graphviz in the initial phase of build the visualization algorithm.

To use *Graphviz* is necessary to make an external call to the system, so we can run the command `dot -Tplain <graph specification>`. The `dot` indicates the location of *Graphviz* in the system (indicated by the users in the plug-in preferences). The `-Tplain` option allow us to get the graph description in a simple text language for the given graph specification. The graph description contain all the necessary information for the build of the visualization algorithm (the CFG elements (nodes and edges), their visual appearance and the most important of all the exact position of each element). The Figure 3.1b shows an example of a graph description.

As the graph description is generated, it is analyzed and stored. The information concerning the elements (as well as their visual appearance) is used to add the elements to the CFG (through the *Zest* API `GraphNode(graph, SWT.NONE, value)` and `GraphConnection(graph, ZestStyles.CONNECTIONS_DIRECTED, start, end)`). The `GraphNode` allow to create a new node with value `value` that will be inserted in the `graph`. Depending on the node type (initial or final) it is defined the corresponding color (green for initial nodes and red for the final nodes) through the `gNode.setBackgroundColor(color)`. Similarly, `GraphConnection` allow

to create a new directed connection between `start` node and `end` node that will be inserted in the `graph`. Once created the nodes, it is made a direct correspondence between the CFG elements and the model elements through the `gNode.setData(modelNode)` (the same happens with the edges). This direct correspondence between CFG elements and the model elements allows us to obtain the information associated with each element in a fast and efficient way, maintaining the separation between the domain and ui (this prevents an unnecessary data stored repetition on both sides (which would be meaningless)).

The information concerning the elements exact position is used to build the visualization algorithm (through `GraphLayoutAlgorithm(parent, elements)` (created by us)). This class receives information about the `elements` (nodes and their positions) and for each node adjusts the viewing scale according to the dimensions of the viewing window (`parent`) and their original position, in order to not deform the CFG. The position of the nodes is set through `nodeEntity.setLocation(pointX, pointY)`, where `nodeEntry` represents the node in the `graph`. The position of the edges is automatically set by *Zest*. The final step in the build is apply the algorithm through `graph.setLayoutAlgorithm(layoutAlgorithm)`. Then the `GraphView` is updated and the CFG is displayed to the user.

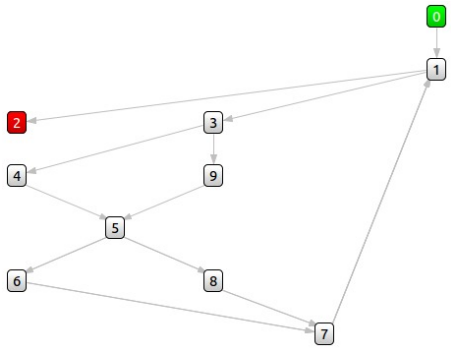


Figure 5.2: CFG generated by *Zest* for the model pictured in Figure 5.1c

5.2 Graph Coverage

This section will present the most important algorithms implemented to generate the set of requirements based on the graphs coverage.

Remember that the generation of the requirements set involves two steps: select the coverage criteria in `Graph Coverage Criteria` view and generate the requirements set for the given criteria in the `Structural Coverage Criteria` view.

We will explain the process of generation of the requirements set from the point the user clicks the generation button in the `Structural Coverage Criteria` view.

The process is quite simple. When the user click the generation button, the first step

is to check what kind of criteria is, if it is a `Structural Coverage Criteria` or a `Data Flow Coverage Criteria`. The reason for this is because `Structural Coverage Criterias` require only the model so they can be generated while the `Data Flow Coverage Criterias` need additional information so they can be generated (we explain in Section 5.2.2 how we get this extra information).

After that check the controller identifies the selected criteria and the execution continues in the class that generates the requirements set for the selected criteria.

5.2.1 Structural Coverage Criteria

Prime Path Coverage

Before explaining the algorithm is necessary to recall the Definitions 2.10 and 2.11 and the Criterion 2.4 (in order to introduce all the concepts used).

The *Prime Path Coverage* is set at the expense of two concepts:

- *Simple path*.
- *Prime path*.

As the Definition 2.10 is introduced, we can break it into two small concepts: a simple path and a cycle path. Formally, the concepts can be defined as follows:

Simple Path: Path from n_i to n_j is simple if no node appears more than once in the path.

Cycle Path: Simple path from n_i to n_j is a cycle if $n_i = n_j$.

We made this separation because it allow us to keep the structures used as simple as possible (which facilitates and simplifies the implementation and makes the code more readable).

In order to generate the requirements set for the prime path coverage criterion it was created a visitor called `SimplePathCoverageVisitor`. For each node in the model (the `currentNode`), the visitor traverses the model in a depth first way. To stored the nodes during the visit we use an auxiliary list (called `pathNodes`) and to stored the requirements set we use a set (called `primePaths`).

The visitor begins the visit in the `currentNode`. It is checked if the `pathNodes` contains the `currentNode`. If it contains, it is checked if the `currentNode` is equal to the first node in the `pathNodes`. If they are equal, the `currentNode` is added to the end of the `pathNodes`. The `pathNodes` is transformed in a `cyclePath`. Then if the `cyclePath` is not a sub-path of any of the other paths already stored in `primePath`, it is added to it, otherwise nothing happens. The last node in the `pathNodes` is removed. The visitor ends the visit to the `currentNode` and continues to the next node in the model from the parent node of the `currentNode`. The `currentNode` is updated with the new node and the algorithm is repeated. If the `currentNode` is different to the first node in the `pathNodes`, the `pathNodes` is transformed in a `simplePath`.

Then if the `simplePath` is not a sub-path of any of the other paths already stored in `primePath`, it is added to it, otherwise nothing happens. The visitor ends the visit to the `currentNode` and continues to the next node in the model from the parent node of the `currentNode`. The `currentNode` is updated with the new node and the algorithm is repeated. If `pathNodes` don't contains the `currentNode`, nothing happens. The `currentNode` is added to the end of the `pathNodes`. Then is checked if the `currentNode` is a final node in the model. If it is, the `pathNodes` is transformed in a `simplePath`. Then if the `simplePath` is not a sub-path of any of the other paths already stored in `primePath`, it is added to it, otherwise nothing happens. The visitor suspends the visit to the `currentNode` and continues to the next node in the model from the `currentNode`. The `currentNode` is updated with the new node and the algorithm is repeated. Each time a node in the model is used the `pathNodes` is clean (not during the visit) and each time a node ends the visit it is removed from the `pathNodes` (only if it had been visited previously, i.e. if it already gone down in the model).

Now that we have the theoretical description of the algorithm, it will be presented their operation, for the CFG shown in Figure 5.2. The total list of prime paths is 41 paths (because it is a very big list, it only be shown the really important parts of the algorithm).

The model has ten nodes ($[0..9]$), the algorithm must be executed for each one (from 0 to 9). The algorithm start in *node 0*, this means that the visitor begins the visit in the *node 0*. `currentNode` $\Rightarrow$ *node 0*. It is checked if the `pathNodes` contains the `currentNode`. The `pathNodes` is empty, so *node 0* is added to it (in `pathNodes` we refer to the nodes only for their value in order to facilitate the understanding, like `pathNodes` (<*node_numbers*>)). Then is checked if `currentNode` is a final node, the answer is no (the only final node is *node 2*), so nothing happens. The visit for *node 0* is suspended, and the visitor goes down in the model. The next node is *node 1*. `currentNode` $\Rightarrow$ *node 1*. It is checked if the `pathNodes` contains the `currentNode`. The `pathNodes` (0) not contains the `currentNode`, so *node 1* is added to it. Then is checked if `currentNode` is a final node, once again the answer is no (nothing happens). The visit for *node 1* is suspended, and the visitor goes down in the model. The next node is *node 2*. `currentNode` $\Rightarrow$ *node 2*. It is checked if the `pathNodes` contains the `currentNode`. The `pathNodes` (0, 1) not contains the `currentNode`, so *node 2* is added to it. Then is checked if `currentNode` is a final node, this time the answer is yes. The `pathNodes` is transformed in the `simplePath` [0, 1, 2] and is checked if [0, 1, 2] is a sub-path of any of the other paths stored in `primePath`. Since `primePath` is empty, [0, 1, 2] is added to it. *Node 2* is a leaf in the model (in other words have no children) which means that the visitor cannot continue going down in the model. *Node 2* ends its visit (2 is removed from `pathNodes`). The visitor returns to *node 1* and continues from the next node, *node 3*. The process for the *nodes 3, 4, 5, 6, 7* is the same as described for *node 0* (with the difference that `pathNodes` have the nodes that have been

visited so far). The difference occurs when the visitor returns to visit *node 1* (coming from *node 7*). `currentNode` ⇒ *node 1*. It is checked if the `pathNodes` contains the `currentNode`. The `pathNodes` (0, 1, 3, 4, 5, 6, 7) contains the `currentNode`. It is checked if the `currentNode` is equal to the first node in the `pathNodes`, the answer for this is no. The `pathNodes` is transformed in the `simplePath` [0, 1, 3, 4, 5, 6, 7]. Then is checked if [0, 1, 3, 4, 5, 6, 7] is a sub-path of any of the other paths stored in `primePath`. Since `primePath` only have [0, 1, 2], [0, 1, 3, 4, 5, 6, 7] is added to it. The visit for *node 1* ends, in this case no node is removed from the `pathNodes`, because *node 1* has not been previously visited. The visitor returns to *node 7* and continues from the next node. Since *node 7* has no more children, it ends its visit, which means 7 is removed from `pathNodes`. The visitor returns to *node 6* and the same happen. The visitor returns to *node 5* and continues from the next node, *node 8*. The visit through the model continues as already described until the moment the visitor ends visit to the *node 0*.

At this point was discovered the following prime paths: [0, 1, 2], [0, 1, 3, 4, 5, 6, 7], [0, 1, 3, 4, 5, 8, 7], [0, 1, 3, 9, 5, 6, 7], [0, 1, 3, 9, 5, 8, 7]. These paths are relative to the *node 0*.

The next node in the model is *node 1*. The `pathNodes` is cleaned. The visitor begins the visit in the *node 1*. `currentNode` ⇒ *node 1*. It is checked if the `pathNodes` contains the `currentNode`. The `pathNodes` is empty, so *node 1* is added to it. Then is checked if `currentNode` is a final node, the answer is no (nothing happens). The visit for *node 1* is suspended, and the visitor goes down in the model. The next node is *node 3*. The process for the *nodes 3, 4, 5, 6, 7* is the same as described for *node 1* (with the difference that `pathNodes` have the nodes that have been visited so far). When the visit of *node 7* is suspended, the `pathNodes` contains the *nodes 1, 3, 4, 5, 6, 7* and the next node to be visited is *node 1* (coming from *node 7*). `currentNode` ⇒ *node 1*. It is checked if the `pathNodes` contains the `currentNode`, the answer for this is yes. Then is checked if the `currentNode` is equal to the first node in the `pathNodes`. The answer for this is also yes, the `currentNode` is added to the `pathNodes`. The `pathNodes` is transformed in the `cyclePath` [1, 3, 4, 5, 6, 7, 1]. Then is checked if [1, 3, 4, 5, 6, 7, 1] is a sub-path of any of the other paths stored in `primePath`, the answer is no, so [1, 3, 4, 5, 6, 7, 1] is added to it. The last node in the `pathNodes` is removed. The visitor ends the visit to the `currentNode`. The visitor returns to *node 7* and continues from the next node. Since *node 7* has no more children, it ends its visit, which means 7 is removed from `pathNodes`. The visitor returns to *node 6* and the same happen. The visitor returns to *node 5* and continues from the next node, *node 8*. The visit through the model continues as already described until the moment the visitor ends visit to the *node 1*. Then the process is repeated for the remaining nodes of the model. At the end all the prime paths are stored in `primePath`.

Complete Path Coverage

Before explaining the algorithm is necessary to recall the Definition 2.7.

As mentioned, *Complete Path Coverage* is not feasible for graphs with cycles; since this results in an infinite number of paths, and hence an infinite number of test requirements.

In order to show as many requirements as possible (without misrepresenting the truth of them), we set the *Complete Path Coverage* at the expense of three concepts

- *Simple path*.
- *Cycle path*.
- *Infinite path*.

The first two concepts are described in the previous section, the third concept (as the name indicates) intended to represent the infinite paths. We consider that a path is infinite if it has cycle, however to not have the same path with a repetition of cycles (which result in an infinite number of paths), we create the base path and add ", ..." to it, indicating that is an infinite path. As show in Listening 5.2.

```
1 [0, 1, 2, 1, 3]
2 [0, 1, 2, 1, 2, 1, 3]          ⟹        [0, 1, 2, 1, ..., 1, 3]
3 [0, 1, 2, 1, 2, 1, 2, 1, 3]
4              ⋮
```

Listing 5.2: Example of how we represent the complete paths.

Recall that we can have cycles within cycles, or more than one path within the cycle, in both cases we represent only the outer cycle, the rest is represented by ", ..." (only in the case of cycles start and end in the same node). Another point that should not be forgotten is that a complete path is a path that starts in a initial node and ends in a final node.

In order to generate the requirements set for the complete path coverage criterion it was created a visitor called `CompletePathCoverageVisitor`, that traverses the model in a depth first way. To stored the nodes during the visit we use an auxiliary list (called `pathNodes`), a stack used to store the cycles (called `stack`) and to stored the requirements set we use a set (called `completePaths`). The actual node is called `currentNode` and the actual cycle is called `currentCycle`.

The visitor begins the visit in the `currentNode`. The cycle at the top of the `stack` is obtained and is stored in `currentCycle`. It is checked if the `currentCycle` contains the `currentNode`. If it contains, the visitor ends the visit to the `currentNode` and continues to the next node in the model from the parent node of the `currentNode`. The `currentNode` is updated with the new node and the algorithm is repeated. Otherwise nothing happens. Then is checked if the `pathNodes` contains the `currentNode` and if the `currentNode` is not a initial node in the model. If and only if both conditions

are met, a `cyclePath` to the sublist from the last occurrence of the `currentNode` to the last node in `pathNodes` (recall that a sublist from *x* to *y* corresponds to the list from *x* to *y - 1*)) is created and is added to the `stack`, otherwise nothing happens. The `currentNode` is added to the end of the `pathNodes`. Then is checked if the `currentNode` is a final node in the model. If it is, the `pathNodes` is transformed in a path (process explain below) and it is added to the `completePaths`, otherwise nothing happens. The visitor suspends the visit to the `currentNode` and continues to the next node in the model from the `currentNode`. The `currentNode` is updated with the new node and the algorithm is repeated. Each time a node ends the visit it is removed from the `pathNodes` (only if it had been visited previously, i.e. if it already gone down in the model), and if the cycle that is at the top of the `stack` starts in this node, it is removed from `stack`, otherwise nothing happens.

Now we explain how we create the paths that are added to the `completePaths`. Remember that to create the path we must have a list of nodes. The first thing we do is check whether the `pathNodes` contains cycles (i.e. if it have repeated nodes). If it don't contains cycles, is created a `simplePath` with the nodes stored in the `pathNodes`. The algorithm ends and the `simplePath` is returned). If it contains cycles, are identified the initial and the final positions (`cycleStart` and `cycleEnd`) of the node that forms the cycle (in the `pathNodes` by their index). Then is checked whether the cycle begins in the first node of the `pathNodes` (the node at index 0). If it not begins, a `simplePath` that represents the `preCycle` is created (the sublist from the node in the index 0 to the node in the index indicated by the `cycleStart` in the `pathNodes`, otherwise nothing happens. Thereafter is check whether the `pathNodes` sublist from `cycleStart` + 1 to `cycleEnd` contain cycles. If the sublist contains cycles, are identified the initial and the final positions (`innerCycleStart` and `innerCycleEnd`) of the node that forms the inner cycle (in the `pathNodes` by their index). The `innerCycleStart` corresponds to the `innerCycleStart` (the position of the node where the inner cycle begins) plus the (`cycleStart` + 1) (this represents the absolute index in the `pathNodes`). Then is created a `infinitePath`. After this, two things happens. First, is created a `cyclePath` to the sublist from the node in the index indicated by the `cycleStart` to the node in the index indicated by the `innerCycleStart` in the `pathNodes`. This `cyclePath` is added to the `infinitePath`. The second thing, the algorithm (of create paths) is repeated for the sublist from the node in the index indicated by the `innerCycleStart` to the node in the index indicated by the `innerCycleEnd` + 1 in the `pathNodes`. This will identify other cycles (if any exist). The result path is added to the `infinitePath`. Then if `innerCycleEnd` + 1 is less than the `cycleEnd`, the algorithm (of create paths) is repeated for the sublist from the node in the index `innerCycleEnd` + 1 to the node in the index `cycleEnd` + 1 in the `pathNodes` (this will check the rest of the nodes) and the result path is added to the `infinitePath`,

otherwise nothing happens. This `infinitePath` represents the `innerPath`. If the sublist from the node in the index indicated by the `cycleStart` + 1 to the node in the index indicated by the `cycleEnd` don't contains cycles, is created a `cyclePath` to the sublist from the node in the index `cycleStart` to the node in the index `cycleEnd` + 1. This `cyclePath` represents the `innerPath`. Then if `cycleEnd` + 1 is less than the `pathNodes` size, the algorithm (of create paths) is repeated for the sublist from the node in the index `cycleEnd` + 1 to the node in the index `pathNodes` size (this will check the rest of the nodes). The result path represents the `posCycle`.

The final step is to assemble the different parts of the paths. If we don't have whether `preCycle` and `posCycle`, The algorithm ends and the `innerPath` is returned. If we have only one of them (`preCycle` or `posCycle`), the algorithm ends and the `preCycle` + `innerPath` or `innerPath` + `posCycle` is returned (depending on the case). If we have both (`preCycle` and `posCycle`), the algorithm ends and the `preCycle` + `innerPath` + `posCycle` is returned. Note that with this description we just wants to give an idea of how the paths are obtained.

Now that we have the theoretical description of the algorithm, it will be presented their operation, for the CFG shown in Figure 5.2. The total list of complete paths is 5 paths (because the process to get the paths is similar to four of them, it only be shown the really important parts of the algorithm).

The cycle at the top of the `stack` is obtained and is stored in `currentCycle`, since the `stack` is empty, the `currentCycle` is *null*. Then is checked if the `currentCycle` contains the `currentNode`, the answer for this is no, so nothing happens. Thereafter is checked if the `pathNodes` contains the `currentNode` and if the `currentNode` is not a initial node in the model. The `pathNodes` is empty, this means that it not contains the `currentNode`, so nothing happens (the second condition don't need to be checked because we are talking about a logical AND), the algorithm proceeds. The `currentNode` is added to the end of the `pathNodes` (in `pathNodes` we refer to the nodes only for their value in order to facilitate the understanding, like `pathNodes` (<*node_numbers*>)). Then is checked if the `currentNode` is a final node in the model, the answer for this is also no, so once again nothing happens. The visit for *node 0* is suspended, and the visitor goes down in the model. The next node is *node 1*. `currentNode` ⇒ *node 1*. The cycle at the top of the `stack` is obtained and is stored in `currentCycle`, since the `stack` is empty, the `currentCycle` is *null*. Thereafter, is checked if the `currentCycle` contains the `currentNode`, the answer for this is no, so nothing happens. Then is checked if the `pathNodes` contains the `currentNode` and if the `currentNode` is not a initial node in the model. Since the `pathNodes` (0) not contains the `currentNode` not contains the `currentNode` nothing happens, the algorithm proceeds. The `currentNode` is added to the end of the `pathNodes`. Then is checked if the `currentNode` is a final node in the model, the answer for this is also

no, so nothing happens. The visit for *node 1* is suspended, and the visitor goes down in the model. The next node is *node 2*. `currentNode` ⇒ *node 2*. The cycle at the top of the `stack` is obtained and is stored in `currentCycle`, since the `stack` is empty, the `currentCycle` is *null*. Then is checked if the `currentCycle` contains the `currentNode`, the answer for this is no, so nothing happens. Thereafter is checked if the `pathNodes` contains the `currentNode` and if the `currentNode` is not a initial node in the model. Since the `pathNodes` (0, 1) not contains the `currentNode` nothing happens, the algorithm proceeds. The `currentNode` is added to the end of the `pathNodes`. Then is checked if the `currentNode` is a final node in the model, the answer for this is yes. The `pathNodes` is passed to the creation paths algorithm. The `pathNodes` don't have cycles. Is created the `simplePath` [0, 1, 2] and it is added to the `completePaths`. *Node 2* is a leaf in the model (in other words have no children) which means that the visitor cannot continue going down in the model. *Node 2* ends its visit, which means 2 is removed from `pathNodes`, and since `stack` is empty nothing else happens. The visitor returns to *node 1* and continues from the next node, *node 3*. The process for the *nodes 3, 4, 5, 6, 7* is the same as described for *node 1* (with the difference that `pathNodes` have the nodes that have been visited so far). When the visit of *node 7* is suspended, the `pathNodes` contains the *nodes 0, 1, 3, 4, 5, 6, 7* and the next node to be visited is *node 1* (coming from *node 7*). `currentNode` ⇒ *node 1*. The cycle at the top of the `stack` is obtained and is stored in `currentCycle`, since the `stack` is empty, the `currentCycle` is *null*. Then is checked if the `currentCycle` contains the `currentNode`, the answer for this is no, so nothing happens. Thereafter is checked if the `pathNodes` (0, 1, 3, 4, 5, 6, 7) contains the `currentNode` and if the `currentNode` is not a initial node in the model, the answer for this is yes. The `cyclePath` [1, 3, 4, 5, 6, 7] (the sublist from the last occurrence of the `currentNode` (1) to the last node (7) (corresponding to the `pathNodes` size) in `pathNodes`) is created and it is added to the `stack`. The `currentNode` is added to the end of the `pathNodes`. Then is checked if the `currentNode` is a final node in the model, the answer for this is no, so nothing happens. The visit for *node 1* is suspended, and the visitor goes down in the model. The next node is *node 2*. `currentNode` ⇒ *node 2*. The cycle at the top of the `stack` is obtained and is stored in `currentCycle`. `currentCycle` ⇒ [1, 3, 4, 5, 6, 7]. Then is checked if the `currentCycle` contains the `currentNode`, the answer for this is no, so nothing happens. Then is checked if the `pathNodes` contains the `currentNode` and if the `currentNode` is not a initial node in the model. Since the `pathNodes` (0, 1, 3, 4, 5, 6, 7, 1) not contains the `currentNode` nothing happens (the second condition don't need to be checked, because we are talking about a logical AND), the algorithm proceeds. The `currentNode` is added to the end of the `pathNodes`. Then is checked if the `currentNode` is a final node in the model, the answer for this is yes. The `pathNodes` is passed to the

creation paths algorithm. Is checked if `pathNodes` have cycles, the answer for this is yes. `cycleStart` ⇒ *1* and `cycleEnd` ⇒ *7*. Then is checked whether the cycle begins in the first node of the `pathNodes`, the answer for this is no. The `simplePath` [0] is created (to the sublist from the node in the index 0 to the node in the index indicated by the `cycleStart` (1) in the `pathNodes`). [0] represents the `preCycle`. Then is check if the `pathNodes` [3, 4, 5, 6, 7] (to the sublist from the node in the index `cycleStart` + 1 (2) to the node in the index indicated by the `cycleEnd` (7) in the `pathNodes`) contain cycles, the answer is no, this means that is created the `cyclePath` [1, 3, 4, 5, 6, 7, 1] (to the sublist from the node in the index `cycleStart` (1) to the node in the index `cycleEnd` + 1 (8)). [1, 3, 4, 5, 6, 7, 1] represents the `innerPath`. Then is checked if `cycleEnd` + 1 is less than the `pathNodes` size, the answer is yes ($8 < 9$), which means that the algorithm (of create paths) is repeated for the sublist [2] (from the node in the index `cycleEnd` + 1 (8) to the node in the index `pathNodes` size (9, because `pathNodes` have the *nodes 0, 1, 3, 4, 5, 6, 7, 1, 2*). The `simplePath` [2] is created. [2] represents the `posCycle`. The final step is to assemble the different parts of the paths. Once we have the `preCycle` and the `posCycle` we have something like `preCycle` + `innerPath` + `posCycle`, this corresponds to the path [0, 1, 3, 4, 5, 6, 7, 1, ..., 1, 2]. [0, 1, 3, 4, 5, 6, 7, 1, ..., 1, 2] is added to the completePaths. The execution continues. *Node 2* is a leaf in the model (in other words have no children) which means that the visitor cannot continue going down in the model. *Node 2* ends its visit, which means 2 is removed from `pathNodes`, and since `stack` have only the cycle [1, 3, 4, 5, 6, 7] (that does not begin with 2), nothing happens. The visitor returns to *node 1* and continues from the next node, *node 3*. `currentNode` ⇒ *node 3*. The cycle at the top of the `stack` is obtained and is stored in `currentCycle`. `currentCycle` ⇒ [1, 3, 4, 5, 6, 7]. Then is checked if the `currentCycle` contains the `currentNode`, the answer is yes. The visitor ends the visit to the `currentNode`. The visitor returns to *node 1* and continues from the next node. Since *node 1* has no more children (already visited *node 2* and *node 3*), it ends its visit, which means 1 is removed from `pathNodes` and once the cycle at the top of the `stack` ([1, 3, 4, 5, 6, 7]) begins with 1, it is removed from the `stack`. The visitor returns to *node 7* and continues from the next node. Since *node 7* has no more children, it ends its visit, which means 7 is removed from `pathNodes`, nothing more happen because *stack* is empty. The visitor returns to *node 6* and the same happen. The visitor returns to *node 5* and continues from the next node, *node 8*. The visit through the model continues as already described until the moment the visitor ends visit to the *node 0*. At the end all the complete paths are stored in `completePath`.

5.2.2 Data Flow Coverage Criteria

All-du-Paths Coverage

Before explaining the algorithm implemented to generate the set of requirements for all-du-paths coverage criterion, it is necessary first explain how we obtain the variables definitions (defs) and its uses.

The process for obtaining the variables defs and its uses is similar to that outlined in Section 5.1.1, based on an *ASTVisitor*.

For this purpose have been implemented two visitors: one for traverse the model called `DefUsesVisitor` and another to traverse the *AST*, called `DefUsesCollector` (an *ASTVisitor*).

The `DefUsesVisitor` traverses the model in a depth first way and as the visitor goes down in the model, the instructions stored on each node (in the graph layer `INSTRUCTIONS`) are accessed. These instructions are portions of the main *AST*, but it remains possible traverse them using an *ASTVisitor* (reason why we did not need to traverse the main *AST* all over again). Before looking in detail to the `DefUsesCollector`, it is necessary first present the structures that were used. To store the variables definitions and uses that occur in the current node were use two sets, called `defs` and `uses` respectively. Each time a new node is visited these sets are cleaned. Another aspect that should be noted is that during the visit exists the possibility that a node could be visited several times, despite this situation the node that begins the visit is added to a set called `visitedNodes` this procedure allow to calculated the set of defs and uses for each node only once.

The `DefUsesCollector` also traverses the *AST* instructions in a depth first way and adds the variables to the `defs` and `uses` as these have been defined or used respectively. In order to have a flexible algorithm that allowed to cover the largest possible number of combinations which can be used between variables (the different possible ways to define a variable (from a simple definition to an array definition) as well as the different possible ways to use a variable (from a simple use to an array access), not forgetting mention the complex things that can be done combining the definitions and the uses (e.g. a variable definition within an array access)), the algorithm is created in a compositional way, this means that as the variables will be found (as the visit to the instructions will advancing) is being inferred if the variables were defined or if they were used. The principle used was visiting first the basic nodes, corresponding to the leaves of the tree (which implies going down everything first) and in the return (i.e. when we go up in the tree) is inferred if the occurrence represents a def or a use. To stored the variables during the visit is uses a stack called `stack`. This algorithm beyond to be applied to the nodes also is applied to edges.

Now we shows the implemented functions and hence the interesting Java elements:

endVisit(StringLiteral node)

When a string literal ends the visit, we add *null* to the `stack`, because a string literal is a constant and we are only interested in represent variables, also this is done to prevent errors when the parent node access the `stack` to inferred the results.

endVisit(CharacterLiteral node)

When a character literal ends the visit, we add *null* to the `stack`, because a character literal is a constant and we are only interested in represent variables, also this is done to prevent errors when the parent node access the `stack` to inferred the results.

endVisit(NumberLiteral node)

When a number literal ends the visit, we add *null* to the `stack`, because a number literal is a constant and we are only interested in represent variables, also this is done to prevent errors when the parent node access the `stack` to inferred the results.

endVisit(BooleanLiteral node)

When a boolean literal ends the visit, we add *null* to the `stack`, because a boolean literal is a constant and we are only interested in represent variables, also this is done to prevent errors when the parent node access the `stack` to inferred the results.

visit(QualifiedName node)

A qualified name is an object composed of two parts: the qualified name and the name. These objects are shown as follow: *x.y*, where the *x* represents the object qualified name and the *y* represents the object name.

When a qualified name is encountered the first thing done is check if the *node* (i.e. the qualified name object) name is a variable. If it isn't a variable, nothing happens. If the *node* name is a variable, the *Java* elements associated with it is obtained (used to check the type of the element (a field, a class file, a initializer, ...)). If the *Java* elements are *null*, the *node* qualified name is added to the `uses` and the *node* qualified name + "." + *node* name is added to the `stack`. This is done because whenever a qualified name is encountered something like *x.y* occurs, which means that *y* is invoked but to do this we first use *x* (which explains why the *node* qualified name (corresponding to the *x*) is added to the `uses`). However at this point, it is impossible to know if the *y* corresponds to a definition or to a use (there isn't enough information to decide), reason why the *node* qualified name + "." + *node* name (corresponding to the *x.y*) is added to the `stack` to be decided later.This notation is used to distinguish the object (e.g if we only have *y* we cannot distinguish *x.y* of *z.y* and the results would be quite confused precluding a clear understanding). If the *Java* elements are not *null*, is checked if it correspond to a field.

If it isn't a field, the *node* name is added to the `stack` to be decided later, this is done because there isn't enough information to decide if it corresponds to a definition or to a use. The information about the *node* qualified name isn't stored because when it isn't a field this information is irrelevant. If it is a field, is checked if the *node* qualified name is a `Type` (e.g. if it is a `enum` class). If it isn't a `Type` nothing happens, otherwise is checked if the *node* qualified name is also a field. If it is, *this* + "." + *node* is added to the `uses`, this is done because the *node* is a internal class field. If the *node* qualified name isn't a field, the *node* qualified name is added to the `uses` and the *node* qualified name + "." + *node* name to the `stack`, to be decided later. At the end, the visitor ends the visit to the *node* and returns to the parent node of the *node*. The visit continues to the next node in the *AST* (if exist, otherwise the data stored in the `stack` are inferred and added to the respective set `defs` or `uses`).

endVisit(SimpleName node)

A simple name is an object composed of only one part, the name, these objects are shown as follow: *x*. These objects represent several elements including variables and constants.

When a simple name ends the visit, the first thing done is check if the *node* (i.e. the simple name object) is a variable. If it isn't, nothing happens. Otherwise, is checked if the *node* represents a field. If it is a field, *this* + "." + *node* name is added to the `stack` to be decided later (at this point there isn't enough information to decide if it corresponds to a definition or to a use), this is done because the *node* is a internal class field. If the *node* isn't a field, the *node* name is added to the `stack` to be decided later. At the end, the visitor ends the visit to the *node* and returns to the parent node of the *node*. The visit continues to the next node in the *AST* (if exist, otherwise the data stored in the `stack` are inferred and added to the respective set `defs` or `uses`).

endVisit(VariableDeclarationFragment node)

A variable declaration fragment is an object that is shown as follow: *<type> <variable> (= <value>)*.

When a variable declaration fragment ends the visit, contrary to what happens with the previous elements, the information about the variables isn't stored in the `stack`, instead it is inferred if the information stored corresponds to a definition or to a use of the variable. This is done by checking the size of the `stack`. If the `stack` size is greater than one (something like *type variable = value* occurs, which means in the `stack` are stored the *variable* and the *value* (at the top)). With this information can be inferred two things: the first is that the element at the top of the `stack` (*value*) is a use, so the *value* is added to the `uses` and then is removed from the `stack`. The second thing is that the new element at the top of the `stack` (*variable*) is a definition, so the *variable* is added to the `defs` and then is removed from the `stack`. If the `stack` size is not greater than

one (something like *type variable* occurs, which means in the `stack` is stored only the *variable*). With this information can be inferred that the element at the top of the `stack` (*variable*) is a definition, so the *variable* is added to the `defs` and then is removed from the `stack`. At the end, the visitor ends the visit to the *node* and returns to the parent node of the *node*. The visit continues to the next node in the *AST* (if exist, otherwise the data stored in the `stack` are inferred and added to the respective set `defs` or `uses`).

endVisit(Assignment node)

An assignment is an object that is shown as follow: *<variable> operator <variable or value>*.

When an assignment ends the visit, as in variable declaration fragment, is also inferred if the information stored in the `stack` corresponds to a definition or to a use of the variable. An assignment is always made up of three elements: the left-side element, the operator and the right-side element (something like *variable operator variable or value* occurs, which means in the `stack` are stored the *variable* and the *variable or value* (at the top)). With this information can be inferred two things: the first is that the element at the top of the `stack` (*variable or value*) is a use, so the *variable or value* is added to the `uses` and then is remove from the `stack`. At this point it is need to check the *node* operator. Depending on the *node* operator the new element at the top of the `stack` in addition to being a definition can also be a use. Is checked if the operator is equal to = (the only case where the variable at the left-side of the operator isn't a use too). If the operator matches with =, nothing happens, otherwise the element at the top of the `stack` is added to the `uses`. The second thing is that the element at the top of the `stack` (*variable*) is a definition, so the *variable* is added to the `defs` and then is remove from the `stack`. At the end, the visitor ends the visit to the *node* and returns to the parent node of the *node*. The visit continues to the next node in the *AST* (if exist, otherwise the data stored in the `stack` are inferred and added to the respective set `defs` or `uses`).

endVisit(InfixExpression node)

An infix expression is an object that is shown as follow: *<variable or value> operator <variable or value> (operator <variable or value> ...)*.

Like in assignment, when an infix expression ends the visit, is inferred if the information stored in the `stack` corresponds to a definition or to a use of the variable. An infix expression is always made up of three elements: the left-side element, the operator and the right-side element (a simple expression), however an infix expression can be composed of several simple expressions (something like this $((a + b) + c) + d$), each one composed also with three elements (two operands (the left-side and the right-side elements) and one operator). In both cases, just for a single expression $(a + b)$ or for multiple expressions $((a + b) + c) + d$ ...) the strategy is the same, both the left-side and the right-side elements

can only be uses. So the first thing to do is checked if the infix expression have multiple expressions. If it have, is add to the `uses` the total number of right-side expressions - 1 (in order to keep only one simple expression), otherwise nothing happens. Then the last expression (or if it is a simple expression) the left-side and the right-side elements are added to the `uses`. The final step is add the *null* to the `stack`, this is done to prevent errors when the parent node access the `stack` to inferred the results. At the end, the visitor ends the visit to the *node* and returns to the parent node of the *node*. The visit continues to the next node in the *AST* (if exist, otherwise the data stored in the `stack` are inferred and added to the respective set `defs` or `uses`).

endVisit(PostfixExpression node)

A postfix expression is an object that is shown as follow: *x++*.

Like in assignment, when a postfix expression ends the visit, is inferred if the information stored in the `stack` corresponds to a definition or to a use of the variable. A postfix expression can be decomposed in the following expression: $x = x + 1$, this means that the *x* is used (in the right-side of the assignment) and is defined (in the left-side of the assignment). The first thing done is store the value at the top of the `stack` in a local variable. Then the element at the top of the `stack` is added to the `uses` and then is removed from the `stack`. The element in the local variable is added to the `defs`.

endVisit(PrefixExpression node)

A prefix expression is an object that is shown as follow: *++x*.

Like in assignment, when a prefix expression ends the visit, is inferred if the information stored in the `stack` corresponds to a definition or to a use of the variable. A prefix expression can be decomposed in the following expression: $(x = x) + 1$, this means that the *x* is used (in the right-side of the assignment) and is defined (in the left-side of the assignment). The first thing done is store the value at the top of the `stack` in a local variable. Then the element at the top of the `stack` is added to the `uses` and then is removed from the `stack`. The element in the local variable is added to the `defs`.

endVisit(ExpressionStatement node)

When a expression-statement ends the visit, all the elements present in the `stack` are added to the `uses` and then are removed from the `stack`.

endVisit(ArrayCreation node)

When an array creation ends the visit, the element at the top of the `stack` is added to the `uses` and then is removed from the `stack`. The final step is add the *null* to the

`stack`, this is done to prevent errors when the parent node access the `stack` to inferred the results.

endVisit(ArrayInitializer node)

When an array initializer ends the visit, the expression corresponding to the array initializer is obtained (this expression corresponds to part that initialize the array when it is created, e.g. *int x* = $\{a, b, c\}$, the expression corresponds to the $\{a, b, c\}$). The first n elements (where n represents the number of elements present in the expression) at the `stack` are added to the `uses` and then are removed from the `stack`. The final step is add the *null* to the `stack`, this is done to prevent errors when the parent node access the `stack` to inferred the results.

endVisit(ArrayAccess node)

When an array access ends the visit, the element at the top of the `stack` is added to the `uses` and then is removed from the `stack`.

visit(MethodInvocation node)

An method invocation is an object that is shown as follow: *(y.)x((<args>))*.

When method invocation is encountered the expression corresponding to the method invocation is obtained (this expression corresponds to the *(y.)x*). If the expression is *null* nothing happens. Otherwise, the visitor is applied to the method invocation expression elements and the visit continues for this elements. All decisions are made at the nodes below and the result of this decisions are added to the `stack`. This process is then repeated for the method invocation arguments and the decisions made are also added to the `stack`. After visiting all the child nodes (corresponding to the elements present in the method invocation expression and the arguments) the `stack` should not be empty (the results of the decisions made in the child nodes), this elements are added to the `uses` and then they are removed from the `stack`. Then the visit to the method invocation node ends.

visit(IfStatement node)

When an if-statement is encountered the expression corresponding to the if-statement guard is obtained. Then the visitor is applied to the if-statement guard elements and the visit continues for this elements. All decisions are made at the nodes below. After visiting all the child nodes (corresponding to the elements present in the if-statement guard) the visit to the if-statement node ends.

visit(WhileStatement node)

When an while-statement is encountered the expression corresponding to the while-statement guard is obtained. Then the visitor is applied to the while-statement guard elements and the visit continues for this elements. All decisions are made at the nodes below. After visiting all the child nodes (corresponding to the elements present in the while-statement guard) the visit to the while-statement node ends.

visit(DoStatement node)

When an do-statement is encountered the expression corresponding to the while-statement guard is obtained. Then the visitor is applied to the while-statement guard elements and the visit continues for this elements. All decisions are made at the nodes below. After visiting all the child nodes (corresponding to the elements present in the while-statement guard) the visit to the do-statement node ends.

visit(ForStatement node)

When an for-statement is encountered the expression corresponding to the for-statement guard is obtained. Then the visitor is applied to the for-statement guard elements and the visit continues for this elements. All decisions are made at the nodes below. After visiting all the child nodes (corresponding to the elements present in the for-statement guard) the visit to the for-statement node ends.

visit(EnhancedForStatement node)

When a enhanced-for-statement is encountered the expression corresponding to the enhanced-for-statement guard is obtained. Then the visitor is applied to the enhanced-for-statement guard elements and the visit continues for this elements. All decisions are made at the nodes below and the result of this decisions are added to the `stack`. After visiting all the child nodes (corresponding to the elements present in the enhanced-for-statement guard) the `stack` should have one element (the result of the decisions made in the child nodes), this element is added to the `uses` and then is removed from the `stack`. Then the parameter corresponding to the enhanced-for-statement variable (where the current iterator value is stored) is obtained. Then the visitor is applied to the enhanced-for-statement parameter element and the visit continues for this element. All decisions are made at the nodes below and the result of this decisions are added to the `stack`. After visiting all the child nodes (corresponding to the element present in the enhanced-for-statement parameter) the `stack` should have one element (the result of the decisions made in the child nodes), this element is added to the `defs` and then is removed from the `stack`. Then the visit to the enhanced-for-statement node ends.

visit(SwitchStatement node)

When a switch-statement is encountered the expression corresponding to the switch-statement guard is obtained. Then the visitor is applied to the switch-statement guard elements and the visit continues for this elements. All decisions are made at the nodes below and the result of this decisions are added to the `stack`. After visiting all the child nodes (corresponding to the elements present in the switch-statement guard) the `stack` should have one element (the result of the decisions made in the child nodes), this element is added to the `uses` and then is removed from the `stack`. Then the visit to the switch-statement node ends.

visit(SwitchCase node)

When an switch-case-statement is encountered the expression corresponding to the switch-case-statement option is obtained. Then the visitor is applied to the switch-case-statement body elements and the visit continues for this elements. All decisions are made at the nodes below. After visiting all the child nodes (corresponding to the elements present in the switch-case-statement body) the visit to the switch-case-statement node ends.

endVisit(ReturnStatement node)

When a return-statement ends the visit, all the elements present in the `stack` are added to the `uses` and then are removed from the `stack`.

When the visit to the instructions ends (i.e. the visit to the `DefUsesCollector` ends), the execution returns to the `DefUsesVisitor`. If at least the `defs` or `uses` is not empty, the two sets are converted in a list with two sub-list (one for the `defs` and another for the `uses`) then this list is stored as a value in a map where the key is the node. At the end of the visit of all nodes, the map have the defs and the uses occurred on each node.

We shall now illustrate the way the class inferred the defs and uses, through an example using the Listing 3.1 (Appendix 3.2.3).

The algorithm starts creating the sets of `defs` and `uses`. The `DefUsesVisitor` starts the visit at the *node 0*. Is checked if the `visitedNodes` contains the *node 0*. The `visitedNodes` is empty, so the visit continues. The sets of `defs` and `uses` are cleaned and the *node 0* is added to the `visitedNodes`. Then the instructions stored in the graph model (on the `INSTRUCTIONS` layer) for the *node 0* are accessed and for each one the `DefUsesCollector` visitor is applied. *Node 0* has only the instruction *int x = 0*. Recall that the visit is made from the bottom to the top of the tree (in most cases). The first method to be visited is the `endVisit(SimpleName node)` (corresponding to the *x*). Is checked if *x* is a variable, the answer is yes. So is checked

if is a field, this time the answer is no, which means that *x* is added to the `stack` to be decided later. The visit for this node ends and then continues to the next node in the *AST*. This time is visit the method `endVisit(NumberLiteral node)` (corresponding to the *0*). This means that is added *null* to the `stack` (to be decided later). The visit for this node ends and then continues to the next node in the *AST*. This time is visit the method `endVisit(VariableDeclarationFragment node)` (corresponding to the *int x = 0*). Is checked if the `stack` size is greater than one, the answer is yes. The element at the top of the `stack` is added to the `uses` and them is removed from the `stack`. Then the new element at the top of the `stack` is added to the `defs` and them is removed from the `stack`. The `DefUsesCollector` visitor ends its visit and the execution returns to the `DefUsesVisitor`. Is checked if at least the `defs` or `uses` is not empty. The `defs` contains *x* and the `uses` is empty (because when a *null* is "added" to the `defs` or to the `uses` nothing happens). As the set of `defs` is not empty, the `defs` and the `uses` are converted in a list and are added to the map together with the *node 0*. Then `DefUsesVisitor` visit continues to the *edge (0, 1)*. The sets of `defs` and `uses` are cleaned. Since it doesn't have information associated with it (stored in the in the graph model on the `GUARDS` layer) nothing happens and the visitor continues to the next node. Then *node 1* is visited. The sets of `defs` and `uses` are cleaned and the *node 0* is added to the `visitedNodes`. Since *node 1* is the node corresponding to the for-statement (also doesn't have information associated with it (stored in the in the graph model on the `INSTRUCTIONS` layer)) nothing happens and the visitor continues. Next is visit the *edge (1, 2)*. The sets of `defs` and `uses` are cleaned. Then the instructions stored in the graph model (on the `GUARDS` layer) for the *edge (1, 2)* are accessed and for each one the `DefUsesCollector` visitor is applied. *Edge (1, 2)* has only the instruction *x <= 3*. The first method to be visited is the `endVisit(SimpleName node)` (corresponding to the *x*). Is checked if *x* is a variable, the answer is yes. So is checked if is a field, this time the answer is no, which means that *x* is added to the `stack` to be decided later. The visit for this node ends and then continues to the next node in the *AST*. This time is visit the method `endVisit(NumberLiteral node)` (corresponding to the *3*). This means that is added *null* to the `stack` (to be decided later). The visit for this node ends and then continues to the next node in the *AST*. This time is visit the method `endVisit(InfixExpression node)` (corresponding to the *x <= 3*). Is checked if the infix expression have multiple expressions, the answer is no, it is a simple expression so the first two elements of the `stack` are added to the `uses` and them are removed from the `stack`. The `DefUsesCollector` visitor ends its visit and the execution returns to the `DefUsesVisitor`. Is checked if at least the `defs` or `uses` is not empty. The `defs` is empty and the `uses` contains *x*. The `defs` and the `uses` are converted in a list and are added to the map together with the *edge (1, 2)*. Then `DefUsesVisitor` visit continues to the remaining nodes and edges in the same way already explained. At the

end of the visit of all nodes, the map have the defs and the uses occurred on each node.

As we mention in the previous chapter, the defs and uses can be viewed by node/edge (as in Table 5.1) or by variable (as in Table 5.2). Although we may choose the view, the defs and uses are only obtained once (unless explicitly indicated to generate them again), this means that as the defs and uses are being inserted on the map, a second map is automatically created from the first but relative to the variables. The creation of the second map in simultaneous with the first allows the program to be more efficient and therefore faster, which saves computational resources and time to the user (since the data is presented instantaneously regardless of the number of times the views change).

Nodes/Edges	***defs***	**uses**
0	$\{x\}$	$\{\}$
(1, 2)	$\{\}$	$\{x\}$
(1, 3)	$\{\}$	$\{x\}$
(3, 4)	$\{\}$	$\{x\}$
(3, 9)	$\{\}$	$\{x\}$
(5, 6)	$\{\}$	$\{x\}$
(5, 8)	$\{\}$	$\{x\}$
7	$\{x\}$	$\{x\}$

Table 5.1: Definitions and uses for nodes and edges of Listing 3.1

Variables	***defs***	**uses**
x	{0, 7}	{(1, 2), (1, 3), (3, 4), (3, 9), (5, 6), (5, 8), 7}

Table 5.2: Definitions and uses for the variables of Listing 3.1

Now that the variables definitions and its uses are identified, the generation of the requirements set for all-du-paths coverage criterion can be explained.

The generation of the requirements set is based on the map that store the defs and uses relative to the variable.

The algorithm is quite simple, in a simple way, the requirements set are the *def-paths* that represent all the variables that are used in the program, in other words the set of path that begin in a def node and end in a use node for all the variables present in the program (as mentioned in the Section 2.2.2.2).

In order to generate the requirements set for the all-du-paths coverage criterion it is necessary to stored the nodes during the visit, for this purpose is used an auxiliary list (called `pathNodes`), to store the requirements set is used a set (called `allDuPaths`). The actual node is called `currentNode`.

The first step is obtain the `defs` and the `uses` sets of the variable. Then is checked if the `uses` set is empty. If it is, nothing happens, and the algorithm is repeated for the

next variable. Otherwise, for each definition (node or edge) is created a visitor called `SimplePathCoverageVisitor` that traverses the model in a depth first way (the visitor operation is explained below). If the definition corresponds to an edge the visitor is applied to the edge source node. When the visit is done if the use corresponds to an edge, it is only consider that the visited node is a use if it is the destination node of one of the edge in the uses set and the previous node in the pathNodes is equal to the edge source node.

The visitor begins the visit in the `currentNode` (corresponding to the definition node). Is checked if the nodes present in the `pathNodes` form a clear path. A clear path is any path that don't have more than one definition node (unless it is the same). So to check it it is a clear path, the following steps are done, is checked if the `pathNodes` is empty. If it is empty, the path is clear. Otherwise is checked if the `currentNode` is a definition node. If it isn't a definition node, the path is clear. Otherwise is checked if the `currentNode` corresponds to the same definition node (the first node in the `pathNodes`) or if it is a new one. If it is the same node, the path is clear. Otherwise the `currentNode` is added to the end of the `pathNodes`. Then is checked if the `currentNode` also is a use node. If it is, the `pathNodes` is transformed in a `path` and it is added to the `allDuPaths`. The last node of the `pathNodes` is remove and the path is not clear. If `currentNode` isn't a use node, the last node of the `pathNodes` is remove and the path is not clear. If the path isn't a clear path, the visitor ends the visit to the `currentNode` and continues to the next node in the model from the parent node of the `currentNode`. The `currentNode` is updated with the new node and the algorithm is repeated. If the path is a clear path, nothing happens. Then is checked if the `pathNodes` contains the `currentNode`. If it contains, it is checked if the `currentNode` is equal to the first node in the `pathNodes`. If they are equal, the `currentNode` is added to the end of the `pathNodes`. The `pathNodes` is transformed in a `path` and it is added to the `allDuPaths`. The last node in the `pathNodes` is removed. The visitor ends the visit to the `currentNode` and continues to the next node in the model from the parent node of the `currentNode`. The `currentNode` is updated with the new node and the algorithm is repeated. If the `currentNode` is different of the first node in the `pathNodes`, the visitor ends the visit to the `currentNode` and continues to the next node in the model from the parent node of the `currentNode`. The `currentNode` is updated with the new node and the algorithm is repeated. If `pathNodes` don't contains the `currentNode`, nothing happens. The `currentNode` is added to the end of the `pathNodes`. Then is checked if the `currentNode` is a use node. If it isn't a use node, nothing happens. Otherwise the `pathNodes` is transformed in a `path` and it is added to the `allDuPaths`. The visitor suspends the visit to the `currentNode` and continues to the next node in the model from the `currentNode`. The `currentNode` is updated with the new node and the algorithm is repeated. Each time a node ends the visit it is

removed from the `pathNodes` (only if it had been visited previously, i.e. if it already gone down in the model).

Now that we have the theoretical description of the algorithm, it will be presented their operation, for the CFG shown in Figure 5.2 and for the elements present in the Table 5.2. The total list of all-du-paths is 24 paths (because it is a very big list, it only be shown the really important parts of the algorithm).

As can be seen in Table 5.2 the program only have one variable (x). The sets of `defs` and `uses` are obtained. Is checked if the set of the *uses* is empty, The answer for this is no (see the column of *uses* in Table 5.2). The algorithm start in *node 0*, this means that the visitor begins the visit in the *node 0* (because is the first node in the set of `defs`). `currentNode` $\Rightarrow$ *node 0*. Is checked if the nodes present in the `pathNodes` form a clear path, the answer for this is yes because `pathNodes` is empty, which means nothing happens. Then is checked if the `pathNodes` contains the `currentNode`. The `pathNodes` is empty, so *node 0* is added to it (in `pathNodes` we refer to the nodes only for their value in order to facilitate the understanding, like `pathNodes` (<*node_numbers*>)). Is check if `currentNode` is a use node, the answer is no, so nothing happens. The visit for *node 0* is suspended, and the visitor goes down in the model. The next node is *node 1*. `currentNode` $\Rightarrow$ *node 1*. The process for the *node 1* is the same as described for *node 0* (with the difference that `pathNodes` have the nodes that have been visited so far). When the visit of *node 1* is suspended, the `pathNodes` contains the *nodes 0, 1* and the next node to be visited is *node 2*. `currentNode` $\Rightarrow$ *node 2*. Is checked if the nodes present in the `pathNodes` form a clear path, the answer for this is yes because *node 2* is not a definition node, which means nothing happens. Then is checked if the `pathNodes` contains the `currentNode`. The `pathNodes` (0, 1) not contains the `currentNode`, so *node 2* is added to it. Then is check if `currentNode` is a use node, the answer is yes (since there is an edge in the `uses` set that have the source node equal to 1 and the destination node equal to 2). The `pathNodes` is transformed in the `path` [0, 1, 2] and it is added to the `allDuPaths`. *Node 2* is a leaf in the model (in other words have no children) which means that the visitor cannot continue going down in the model. *Node 2* ends its visit (2 is removed from `pathNodes`). The visitor returns to *node 1* and continues from the next node, *node 3*. The process for the *node 3* is the same as described for *node 2*, which means that the `path` [0, 1, 3] is added to the `allDuPaths`. The visit for *node 3* is suspended, and the visitor goes down in the model. The visit through the model continues as already described until the moment the visitor ends visit to the *node 0*. Then all the process is repeat to the next definition node (*node 7*). At the end all the all-du-paths are stored in `allDuPaths`.

5.2.3 Tour Types

Tour

Before explaining the algorithm is necessary to recall the Definitions 2.8. As mentioned, p is said to tour subpath q, if q is a subpath of p, where p is a test path and q is a path from the requirements set.

To check if the q is a subpath of p, is made a verification for every node present in the p to check if it occurs in q in the exactly the same order. The current node of p and the current node of q are compared to each other. If they are different, the current node of p is updated with the next node of p, until it is equal to the current node of q. When the current node of p is equal to the current node of q, the current node of p is updated with the next node of p, as well as the current node of q is updated with the next node of q, and they are compared again. If they are equal this process is repeated (until one of the paths p or q don't have more nodes), which means that q is subpath of p, otherwise the algorithm ends and q is not subpath of p. All the path in the requirements set are checked to the test path.

Let's demonstrate how this is done through an example. Recall the 5 prime paths ([0, 1, 2], [0, 1, 3, 4, 5, 6, 7], [0, 1, 3, 4, 5, 8, 7], [0, 1, 3, 9, 5, 6, 7], [0, 1, 3, 9, 5, 8, 7]) generated for *node 0* (in Section 5.2.1 for the prime path coverage criterion). Suppose that the test path is [0, 1, 3, 4, 5, 6, 7, 1, 2]. In order to facilitate the understanding `currentTestNode` represents the current node of the test path *(p)* and `currentRequirementNode` represents the current node of the requirement path *(q)*.

The first test requirement path is [0, 1, 2], `currentTestNode` ⇒ *node 0* (the first node of the test path), `currentRequirementNode` ⇒ *node 0* (the first node of the requirement path). `currentTestNode` and `currentRequirementNode` are compared to each other to check if they are equal, the answer is yes. The values are updated to the next nodes `currentTestNode` ⇒ *node 1*, `currentRequirementNode` ⇒ *node 1*. They are compared again, they still equals, so the values are updated to the next nodes `currentTestNode` ⇒ *node 3*, `currentRequirementNode` ⇒ *node 2*. They are compared again, but this time they are different, which means that [0, 1, 2] isn't subpath of [0, 1, 3, 4, 5, 6, 7, 1, 2]. The next test requirement path is [0, 1, 3, 4, 5, 6, 7]. `currentTestNode` ⇒ *node 0*, `currentRequirementNode` ⇒ *node 0*. `currentTestNode` and `currentRequirementNode` are compared to each other to check is they are equal, the answer is yes, and the values for both are updated. The process is the same for the remaining nodes. After checking the *node 7*, the `currentRequirementNode` cannot be updated, because [0, 1, 3, 4, 5, 6, 7] don't have more nodes, this means that all the nodes in [0, 1, 3, 4, 5, 6, 7] occur in [0, 1, 3, 4, 5, 6, 7, 1, 2] in the exactly the same order, in other words [0, 1, 3, 4, 5, 6, 7] is subpath of [0, 1, 3, 4, 5, 6, 7, 1, 2].

The remaining test requirement paths like [0, 1, 2] also aren't subpath of [0, 1, 3, 4, 5, 6, 7, 1, 2].

Sidetrip

Before explaining the algorithm is necessary to recall the Definitions 2.12. As mentioned, p is said to tour the subpath q with a sidetrip, if and only if every edge in q is also in p in the same order, where p is a test path and q is a path from the requirements set.

To check if p tours the subpath q with a sidetrip, is made a verification for every node present in the p to check if it occurs in q in the same order. The current node of p and the current node of q are compared to each other. If they are different, the previous node of p is stored (which happens only when the they became different, when they are already different this value is not changed) and the current node of p is updated with the next node of p, this process is repeated until it is equal to the current node of q (or until the p have no more nodes). When the current node of p is equal to the current node of q, is checked if the previous node of the current node of p is equal to the node stored (to ensure that the edge is the correct). If they are equal the current node of p is updated with the next node of p, as well as the current node of q is updated with the next node of q, and they are compared again, otherwise the algorithm is repeated. This process is repeated until one of the paths p or q don't have more nodes. If all the edges present in q are in p in the same order this means that p tours the subpath q with a sidetrip, otherwise p don't tour the subpath q with a sidetrip. All the path in the requirements set are checked to the test path.

Let's demonstrate how this is done through an example (it will be used the same requirement paths and the same test path used to explain the *tour* algorithm). In order to facilitate the understanding `previousTestNode` represents the previous node of the test path *(p)*, `currentTestNode` represents the current node of the test path *(p)* and `currentRequirementNode` represents the current node of the requirement path *(q)*.

The first test requirement path is [0, 1, 2], `currentTestNode` ⇒ *node 0* (the first node of the test path), `currentRequirementNode` ⇒ *node 0* (the first node of the requirement path). `currentTestNode` and `currentRequirementNode` are compared to each other to check if they are equal, the answer is yes. The values are updated to the next nodes `currentTestNode` ⇒ *node 1*, `currentRequirementNode` ⇒ *node 1*. They are compared again, they still equals, so the values are updated to the next nodes `currentTestNode` ⇒ *node 3*, `currentRequirementNode` ⇒ *node 2*. They are compared again, but this time they are different, which means that the previous node of the `currentTestNode` is stored, `previousTestNode` ⇒ *node 1* and `currentTestNode` is updated to the next node, `currentTestNode` ⇒ *node 4*. They are compared again to check if they are equal, the answer is also no. This procedure is the same for *nodes 5, 6, 7, 1*. `currentTestNode` is updated to the next node,

`currentTestNode` ⇒ *node 2*. It is compared with `currentRequirementNode` to check if they are equal, the answer this time is yes. Then is checked if the previous node of the `currentTestNode` (*node 1*) is equal to `previousTestNode`, the answer is yes. After checking the *node 2*, the `currentRequirementNode` cannot be updated, because [0, 1, 2] don't have more nodes, this means that all the edges in [0, 1, 2] occur in [0, 1, 3, 4, 5, 6, 7, 1, 2] in the the same order, in other words [0, 1, 3, 4, 5, 6, 7, 1, 2] tours the subpath [0, 1, 2] with a sidetrip. The same happens for the requirement paths [0, 1, 3, 4, 5, 6, 7]. [0, 1, 3, 4, 5, 6, 7, 1, 2] don't tour [0, 1, 3, 4, 5, 8, 7], [0, 1, 3, 9, 5, 6, 7] or [0, 1, 3, 9, 5, 8, 7] with sidetrip because each one have a node (*node 8* or *node 9*) that doesn't make part of the [0, 1, 3, 4, 5, 6, 7, 1, 2], which means that [0, 1, 3, 4, 5, 6, 7, 1, 2] has no edges that begin or end with *node 8* or *node 9*.

Detour

Before explaining the algorithm is necessary to recall the Definitions 2.13. As mentioned, p is said to tour the subpath q with a detour, if and only if every node in q is also in p in the same order, where p is a test path and q is a path from the requirements set.

To check if p tours the subpath q with a detour, is made a verification for every node present in the p to check if it occurs in q in the same order. The current node of p and the current node of q are compared to each other. If they are different, the current node of p is updated with the next node of p, an they are compared again. When the current node of p is equal to the current node of q, the current node of p is updated with the next node of p, as well as the current node of q is updated with the next node of q, and they are compared again. This process is repeated until one of the paths p or q don't have more nodes. If all the nodes present in q are in p in the same order this means that p tours the subpath q with a detour, otherwise p don't tour the subpath q with a detour. All the path in the requirements set are checked to the test path.

Let's demonstrate how this is done through an example (it will be used the same requirement paths and the same test path used to explain the *tour* algorithm). In order to facilitate the understanding `currentTestNode` represents the current node of the test path *(p)* and `currentRequirementNode` represents the current node of the requirement path *(q)*.

The first test requirement path is [0, 1, 2], `currentTestNode` ⇒ *node 0* (the first node of the test path), `currentRequirementNode` ⇒ *node 0* (the first node of the requirement path). `currentTestNode` and `currentRequirementNode` are compared to each other to check if they are equal, the answer is yes. The values are updated to the next nodes `currentTestNode` ⇒ *node 1*, `currentRequirementNode` ⇒ *node 1*. They are compared again, they still equals, so the values are updated to the next nodes `currentTestNode` ⇒ *node 3*, `currentRequirementNode` ⇒ *node 2*. They are compared again, but this time they are different, which means that the `current-`

`TestNode` is updated to the next node, `currentTestNode` ⇒ *node 4*. They are compared again to check if they are equal, the answer is also no. This procedure is the same for *nodes 5, 6, 7, 1*. `currentTestNode` is updated to the next node, `currentTestNode` ⇒ *node 2*. It is compared with `currentRequirementNode` to check if they are equal, the answer this time is yes. After checking the *node 2*, the `currentRequirementNode` cannot be updated, because [0, 1, 2] don't have more nodes, this means that all the nodes in [0, 1, 2] occur in [0, 1, 3, 4, 5, 6, 7, 1, 2] in the the same order, in other words [0, 1, 3, 4, 5, 6, 7, 1, 2] tours the subpath [0, 1, 2] with a detour. The same happens for the requirement paths [0, 1, 3, 4, 5, 6, 7] and . [0, 1, 3, 4, 5, 6, 7, 1, 2] do not tour [0, 1, 3, 4, 5, 8, 7], [0, 1, 3, 9, 5, 6, 7] or [0, 1, 3, 9, 5, 8, 7] with detour because each one have a node (*node 8* or *node 9*) that doesn't make part of the [0, 1, 3, 4, 5, 6, 7, 1, 2].

5.3 Test Path

5.3.1 Test Path Generation

For generating the test paths first is necessary to have the set of requirements. The pseudocode below explains the implement algorithm:

```
for each requirement path in requirements set
  if requirement path begins in an initial node and ends in a
     final node {
    requirement path is added to the test path set
    requirement path is remove
  }
get first requirement path from the requirement set
pos=0
store all the requirement path nodes in a list
remove the requirement path
pos++
```

The next step is find another requirement path(s) that have the largest number of consecutive nodes equal to a sublist. The compare process uses a variable—`pos`. The sublist correspond to the nodes from `pos` to the last position of the list. The sublist is compared to the first *n* nodes of the requirement path (*n* corresponds to the size of the sublist).

```
for each requirement path in requirements set
  if requirement path has the maximum number of consecutive
     nodes equal to the sublist
    store the requirement path in a list
```

If exists only one requirement path in the list, the nodes that have not been compared are added to the list, `pos` is incremented. If exists several requirement paths in the list, it is

chosen the rightmost—the path that have the nodes with bigger values. Thereafter, the procedure is the same as described for the case that we have only one requirement path. In both cases the last node cannot be a final node (explained below). If not exist any requirement path in the list, `pos` is incremented.

All the requirement paths are only used once, to ensure that this happens, when one requirement path is in the list and it last node is a final node, instead of add the nodes that have not been compared to the list, the entire path is stored (as well as `pos`) and the algorithm continues. Then, if are found others requirement paths in the list, the procedure is the same explained above and the requirement path stored is discard. Otherwise, the remaining nodes of the requirement path stored are added to the list, since the last node is a final node, is created a path with the nodes stored in the list, and it is added to the test path set (the requirement path stored is discard and is marked as used). This decision is made in order to include the largest number of requirement paths in a single test, this allows to create the minimum test paths required to cover all the requirement path (given by the maximum between the number of requirement paths that start in a initial node and the number of requirement paths that end in a final node). If after the requirement paths that begin at the initial node have been marked as used and given rise to test paths, still exist other requirement path that are unused, is obtained the nodes from the initial node to the first node of that requirement path. This nodes are added to the list, as well as the nodes of the requirement path. The algorithm is repeated until all requirement paths are marked as used, each time the algorithm start all over again the list is cleaned and the variable start from the initial position.

Let's demonstrate how this is done through an example. For this example will be use the requirement paths generated for the prime path coverage criterion (see Appendix A (Table A.1)). The algorithm starts checking if exists any requirement path that begins in a initial node and ends in a final node, the answer for this is yes (the requirement path [0, 1, 2]). [0, 1, 2] is added to the test path set and is marked as used. The next requirement path is [0, 1, 3, 4, 5, 6, 7]. `pathNodes` is empty and `pos` $\Rightarrow$ 0. 0, 1, 3, 4, 5, 6, 7 are added to the `pathNodes`. `pos` $\Rightarrow$ 1. [0, 1, 3, 4, 5, 6, 7] is marked as used. All the requirement paths are compared with the nodes in the sublist [1, 3, 4, 5, 6, 7] (corresponding to the nodes from the position indicated by `pos` to the `pathNodes` size) and the requirement path that matches all nodes is [1, 3, 4, 5, 6, 7, 1]. 1 is added to the `pathNodes`. `pos` $\Rightarrow$ 2. [1, 3, 4, 5, 6, 7, 1] is marked as used. Then all the requirement paths are compared with the nodes in the sublist [3, 4, 5, 6, 7, 1]. The requirement paths that matches all nodes are [3, 4, 5, 6, 7, 1, 2] and [3, 4, 5, 6, 7, 1, 3]. 3 is added to the `pathNodes`. `pos` $\Rightarrow$ 3. [3, 4, 5, 6, 7, 1, 3] is marked as used (is choose [3, 4, 5, 6, 7, 1, 3] because it is the rightmost path and also because the last node of [3, 4, 5, 6, 7, 1, 2] is a final node). This process is repeated all over again for the remaining requirement path. The Listening 5.3 shows in a more comprehensive way how the test paths are generated. In order to facilitate

the understanding of the test paths generation process, the nodes that have been compared in each iteration are listed above the same nodes of the previous requirement path (e.g the nodes that have been compared for the requirement paths in the lines 1 and 2 are nodes 1, 3, 4, 5, 6, 7). The nodes that are added to the `pathNodes` are those that aren't in the previous node (e.g the node 1 is the only node added in line 2). The number that are in front of the requirement paths is the requirement path id (indicated in the Table A.1), this is used to mark the requirement paths that were used (when the algorithm ends it is used to check two things if all requirement path were used and if they were used only once). Although exists more than one requirement path that matches all nodes (e.g to match the sublist [3, 4, 5, 6, 7, 1] (line 2) exists two requirement paths [3, 4, 5, 6, 7, 1, 2] and [3, 4, 5, 6, 7, 1, 3]) is shown only the requirement path that is used. When exists an id before the requirement path (e.g as in line 15) this means that the only requirement path that matches all nodes of the previous requirement path, ends with a final node (in this case, for the sublist [3, 9, 5, 8] the only requirement path that matches all this nodes is the requirement path [3, 9, 5, 8, 7, 1, 2], since this requirement path ends with a final node the algorithm continues, which means, [3, 9, 5, 8, 7, 1, 2] is stored and the `pos` moves one position (the sublist is updated to [9, 5, 8]), then the nodes in the sublist are compared with the requirement path that still unused, for this sublist is found another requirement path [9, 5, 8, 7, 1, 3, 4], [3, 9, 5, 8, 7, 1, 2] is discard and nodes 7, 1, 3, 4 are added to the `pathNodes`, the algorithm continues from this point).

```
1  [0, 1, 3, 4, 5, 6, 7] (2)
2     [1, 3, 4, 5, 6, 7, 1] (6)
3        [3, 4, 5, 6, 7, 1, 3] (11)
4           [4, 5, 6, 7, 1, 3, 9] (19)
5              [5, 6, 7, 1, 3, 9, 5] (23)
6                 [6, 7, 1, 3, 9, 5, 8] (29)
7                    [7, 1, 3, 9, 5, 8, 7] (33)
8                       [1, 3, 9, 5, 8, 7, 1] (9)
9                          [3, 9, 5, 8, 7, 1, 3] (17)
10                            [9, 5, 8, 7, 1, 3, 9] (41)
11                               [5, 8, 7, 1, 3, 9, 5] (25)
12                                  [8, 7, 1, 3, 9, 5, 8] (37)
13
14 [8, 7, 1, 3, 9, 5, 8] (37)
15        (16) [9, 5, 8, 7, 1, 3, 4] (40)
16                [5, 8, 7, 1, 3, 4, 5] (24)
17                   [8, 7, 1, 3, 4, 5, 8] (35)
18                      [7, 1, 3, 4, 5, 8, 7] (31)
19                         [1, 3, 4, 5, 8, 7, 1] (7)
20                            [3, 4, 5, 8, 7, 1, 3] (13)
21                               [4, 5, 8, 7, 1, 3, 9] (21)
22                                     [8, 7, 1, 3, 9, 5, 6] (36)
23
24 [8, 7, 1, 3, 9, 5, 6] (36)
```

```
   [7, 1, 3, 9, 5, 6, 7] (32)
      [1, 3, 9, 5, 6, 7, 1] (8)
         [3, 9, 5, 6, 7, 1, 3] (15)
            [9, 5, 6, 7, 1, 3, 9] (39)
               [6, 7, 1, 3, 9, 5, 6] (28)
                  (14) [9, 5, 6, 7, 1, 3, 4] (38)
                     [5, 6, 7, 1, 3, 4, 5] (22)
                        [6, 7, 1, 3, 4, 5, 8] (27)

[6, 7, 1, 3, 4, 5, 8] (27)
       (12) [4, 5, 8, 7, 1, 3, 4] (20)
               [8, 7, 1, 3, 4, 5, 6] (34)
                  [7, 1, 3, 4, 5, 6, 7] (30)
                     (10) [4, 5, 6, 7, 1, 3, 4] (18)
                        [6, 7, 1, 3, 4, 5, 6] (26)

[6, 7, 1, 3, 4, 5, 6] (26)
    (10) [3, 4, 5, 6, 7, 1, 2] (10)
```

Listing 5.3: Process of generate a test path for the requirement paths present in Table A.1.

At the end, is created a path with all nodes stored in `pathNodes` (which corresponds to the path [0, 1, 3, 4, 5, 6, 7, 1, 3, 9, 5, 8, 7, 1, 3, 9, 5, 8, 7, 1, 3, 4, 5, 8, 7, 1, 3, 9, 5, 6, 7, 1, 3, 9, 5, 6, 7, 1, 3, 4, 5, 8, 7, 1, 3, 4, 5, 6, 7, 1, 3, 4, 5, 6, 7, 1, 2]), this path is added to the test path set. After this the algorithm continues. The next requirement path is [0, 1, 3, 4, 5, 8, 7]. The `pathNodes` is cleaned and `pos` $\Rightarrow$ 0. The nodes 0, 1, 3, 4, 5, 8 and 7 are added to the `pathNodes`. `pos` $\Rightarrow$ 1, which corresponds to the sublist [1, 3, 4, 5, 8, 7]. This sublist is compared to the remaining requirement path that still unused. Not exist any requirement path that matches all sublist nodes. `pos` $\Rightarrow$ 2, which corresponds to the sublist [3, 4, 5, 8, 7]. This sublist is compared to the remaining requirement path that still unused. Is found the requirement path [3, 4, 5, 8, 7, 1, 2]. This requirement path ends in a final node, which means that [3, 4, 5, 8, 7, 1, 2] is stored (as the current `pos` value) and the algorithm continues. `pos` $\Rightarrow$ 3, which corresponds to the sublist [4, 5, 8, 7]. Not exist any requirement path that matches all sublist nodes, so the algorithm continues. This process is repeated for the `pos` $\Rightarrow$ 4, 5 and 6. In both cases not exist any requirement path that matches all sublist nodes, which means that the requirement path stored ([3, 4, 5, 8, 7, 1, 2]) is marked as used and the nodes 1 and 2 are added to the `pathNodes`. Since the requirement path ends with a final node, is created a path with all nodes stored in `pathNodes` (which corresponds to the path [0, 1, 3, 4, 5, 8, 7, 1, 2]), this path is added to the test path set. After this the algorithm continues. The process for the remaining requirement paths is similar to this and are created the following path: [0, 1, 3, 9, 5, 6, 7, 1, 2] and [0, 1, 3, 9, 5, 8, 7, 1, 2]. The Listening 5.3.1 shows the process for the remaining test paths.

```
[0, 1, 3, 4, 5, 8, 7] (3)
 (12) [3, 4, 5, 8, 7, 1, 2] (12)

[0, 1, 3, 9, 5, 6, 7] (4)
 (14) [3, 9, 5, 6, 7, 1, 2] (14)

[0, 1, 3, 9, 5, 8, 7] (5)
 (16) [3, 9, 5, 8, 7, 1, 2] (16)
```

As can be seen all the requirement paths are used and are used only once, in other words, with this five test paths we get all requirement paths coverage (full coverage).

5.3.2 Test Path Execution

The test path execution represents the executed path in the CFG resulting for a test run. Launching a test involves several steps (cf., Section 4.3.3), like the creation of the `rules.btm` file and the `output.txt` files. The `rules.btm` file specifies the set of rules that *Byteman* uses to instrument the bytecode, while the `output.txt` file is used to store the trace information collected during the test execution. In order to have an efficient tracing algorithm, rules instrument:

- the method entry;
- the method exit; and
- a few edges.

The algorithm starts by inserting probes at the beginning and at the end of the method. This is done to distinguish the different tests, and insert marks in the `output.txt` file to delimit each test path. Then, to select the edges where the remaining probes will be inserted:

```
for each node in the graph {
  for each edge in graph
    if the edge source node is equal to the node
      the edge is added to the edgeSet
  if node is a condition node
    for each edge in the edgeSet
      if the edge destination node has information
        a rule is created
  else
     for each edge in the edgeSet
        if edge destination node is a condition node or
        if edge destination node has information
          a rule is created
}
```

A condition node—an *if-statement*, or a *while-statement*, or a *do-statement*, or a *for-statement*, or an *enhanced-for-statement*, or a *switch-statement*—this is done checking the information stored in the graph `INSTRUCTIONS` layer.

To create a rule are necessary the following elements: the rule name, the rule line and the edge. To create the rules was adopted the following convention, the rule name is the method name and the line number (e.g `RULE method Line10`), if the rule represents the method entry or exit then the rule name is the method name and entry or exit respectively (e.g `RULE method entry` or `RULE method exit`), each time this rule is executed is added to the `output.txt` file "entry" or "exit" respectively. The rule line represents the line where the rule should be placed (this information is obtained in the graph `INSTRUCTIONS` layer). The edge is the element that will be added to the `output.txt` file if the rule is fired, in other words, if the program execution reaches the line indicated in the rule, the edge(s) are always added to the `output.txt` file, because the condition is always *true* (`IF true DO <add edge to output.txt>`). Before create the rule is checked if already exists a rule for the given line. If not exists, a new rule is created, as shown below.

```
1  RULE rule1
2  CLASS class_name
3  METHOD method_name
4  AT LINE line1
5  IF true
6  DO edge1
7  ENDRULE
8
9  RULE rule2
10 CLASS class_name
11 METHOD method_name
12 AT LINE line2
13 IF true
14 DO edge2
15 ENDRULE
```

If already exists a rule for the given line, instead of create a new rule, only the edge is added to the existing rule, as shown below.

```
1  RULE rule1
2  CLASS class_name
3  METHOD method_name
4  AT LINE line1
5  IF true
6  DO edge1 edge2
7  ENDRULE
```

When all tests finish execution, the `output.txt` file contains all the information associated with each test execution. However, this information needs to be manipulated in order to obtain the execution paths.

The algorithm to obtain the tests execution path consists in reading the `output.txt` file line by line.

```
until not read the end of file (of the output.txt file) {
  read a line
  if it reads "entry" or "exit" {
    mark the respective element
    if "entry" and "exit" are marked
      return test path
  } else if it reads a list of edges {
      get the first edge of the list
      if edge source node is equal to the path last node {
        edge destination node is added to the end of the path
        remove edge
        if exist more edges in the list {
          for each edge in the list
            if edge source node is equal to the path last node {
              edge destination node is added to the end of the
                 path
              remove edge
            }
          if list is not empty
            list is pushed to the stack
        }
      } else {
          get the list of edges at the top of the stack
          boolean isUsed = false;
          while list is not empty and all the edges destination
             node cannot be incorporated in the path and isUsed
             is false {
            for each edge in the list
              if edge source node is equal to the path last node
                  {
                edge destination node is added to the end of the
                    path
                remove edge
                isUsed = true
              }
            if isUsed is false
              list is poped from the stack
          }
          edge destination node is added to the end of the path
          remove edge
          if exist more edges in the list {
            while list is not empty and all the edges
               destination node cannot be incorporated in the
```

```
                path
38              for each edge in the list
39                if edge source node is equal to the path last
                     node {
40                  edge destination node is added to the end of
                       the path
41                  remove edge
42                }
43            if list is not empty
44              list is pushed to the stack
45          }
46        }
47      }
48  }
49  }
```

Although this algorithm has been built in a compositional way (the only way found to be able to overcome the constraint presented by *Byteman* (more details about this in Section 3.2.3)), in certain cases the information stored in the `output.txt` file is not sufficient to obtain the executed path. One of this cases is when the final node do not have instructions (i.e., when the node do not have information associated with it in the graph `INSTRUCTIONS` layer). This means that no rule will be enforced and consequently the `output.txt` file do not contains any edge where the destination node is a final node. For this case the remaining nodes (from the last node in the path to the closest final node) are inferred. This is done by:

```
1  get the list of edges at the top of the stack
2  until there is no more edges that can be incorporated into the
      path {
3    for each edge in the list
4      if edge source node is equal to the path last node {
5        destination node is added to the end of the path
6        remove edge
7    }
8    traverses the graph in a deep first way from the last node in
        the path {
9      added node to the end of te path
10     if is a final node
11       break
12   }
13 }
```

Let us demonstrate the algorithm by means of an example. Recall the method `demo` shown in the Listing 3.1. When the user clicks in the `Run As` ⇒ `PESTT - JUnit`

`test` button to run the test `rules.btm` and `output.txt` files are created. The instrumentation rules (`rules.btm`) are:

```
RULE demo entry
CLASS DemoClass
METHOD demo
HELPER domain.tests.instrument.HelperClass
AT ENTRY
IF true
DO debug("Entering in method demo")
ENDRULE

RULE demo exit
CLASS DemoClass
METHOD demo
HELPER domain.tests.instrument.HelperClass
AT EXIT
IF true
DO debug("Exiting method demo")
ENDRULE

RULE demo Line9
CLASS DemoClass
METHOD demo
HELPER domain.tests.instrument.HelperClass
AT LINE 9
IF true
DO debug("(0, 1) (6, 7) (7, 1) (8, 7)")
ENDRULE

RULE demo Line10
CLASS DemoClass
METHOD demo
HELPER domain.tests.instrument.HelperClass
AT LINE 10
IF true
DO debug("(1, 3)")
ENDRULE

RULE demo Line11
CLASS DemoClass
METHOD demo
HELPER domain.tests.instrument.HelperClass
AT LINE 11
IF true
DO debug("(3, 4) (4, 5)")
ENDRULE

RULE demo Line13
CLASS DemoClass
```

```
48 METHOD demo
49 HELPER domain.tests.instrument.HelperClass
50 AT LINE 13
51 IF true
52 DO debug("(3, 9) (9, 5)")
53 ENDRULE
54
55 RULE demo Line15
56 CLASS DemoClass
57 METHOD demo
58 HELPER domain.tests.instrument.HelperClass
59 AT LINE 15
60 IF true
61 DO debug("(5, 6)")
62 ENDRULE
63
64 RULE demo Line17
65 CLASS DemoClass
66 METHOD demo
67 HELPER domain.tests.instrument.HelperClass
68 AT LINE 17
69 IF true
70 DO debug("(5, 8)")
71 ENDRULE
```

Listing 5.4: Rules used to obtain the executed paths.

Then, the program is executed and each time a rule is reached the respective element ("entry", "exit" or edge) is added to the `output.txt` file. When the program finishes executing, the content of the `output.txt` file is as follows:

```
1  Entering in method demo
2  (0, 1) (6, 7) (7, 1) (8, 7)
3  (1, 3)
4  (3, 4) (4, 5)
5  (5, 8)
6  (1, 3)
7  (3, 9) (9, 5)
8  (5, 8)
9  (1, 3)
10 (3, 4) (4, 5)
11 (5, 6)
12 (1, 3)
13 (3, 9) (9, 5)
14 (5, 6)
15 Exiting method demo
```

Listing 5.5: Content of the `output.txt` generated by running the test shown in Listing 3.2 with the rules shown in Listing 5.4.

In order to facilitate the understanding of the demonstration, we use a structure to store the path nodes (`pathNodes`) and a `stack` to store the remaining edges (as list of edges). Since we only have executed a test, we only have a test path as marked in line 1 and line 15. Line 2 is read. Since the `pathNodes` is empty, it is checked which edge has a source node that is an initial node. The answer is (0, 1), so *nodes 0* and *1* are added to the `pathNodes`. Then, it is checked if exist other edges that can be incorporated directly to the path (only those that has not been used in the current line), the answer is no. The remaining edges are stored in the `stack`. Is read the next line. (1, 3) can be incorporated directly to the `pathNodes`, which means *node 3* is added to `pathNodes` (because the last node in `pathNodes` is *node 1*, as well as the edge source node). The line don't have more edges, so the next line is read. The previous procedure also applies to the edges (3, 4), (4, 5) and (5, 8). At this point the `pathNodes` contains the *nodes 0, 1, 3, 4, 5, 8*. Then is read the next line. However in this case the edge (1, 3) cannot be incorporate directly to the `pathNodes` (because the last node in `pathNodes` is *node 8* and the edge source node is *node 1*). The edges at the top of the `stack` are used, this means that is check if exists an edge that can be incorporated directly to the path. The edge (8, 7) can be incorporated directly to the path, so *node 7* is added to the `pathNodes`. Then is checked if exists other edge that can be incorporated directly to the path. The edge (7, 1) can be, *node 1* is added to the `pathNodes`. The procedure is repeated, but this time none of the edges can be incorporated directly in the path. Then is checked again if the edge (1, 3) can now be incorporated to the path. The answer is yes so *node 3* is added to the `pathNodes`. The algorithm continues in the same way as explained. At the end the `pathNodes` contains the following sequence of nodes: 0, 1, 3, 4, 5, 8, 7, 1, 3, 9, 5, 8, 7, 1, 3, 4, 5, 6, 7, 1, 3, 9, 5, 6. With this nodes a test path cannot be created (because the last node is not a final node). Is checked if exist edges at the top of the `stack` that can be used. The answer for this is yes, so *node 7* and *node 1* are added to the `pathNodes`. Even with this, the path cannot be created (the last node still do not is a final node). The las step is inferred if the remaining nodes, i.e. traverse the graph in a depth first way from the last node in the path until a final node be found. The traverse begins in *node 1*. The first node visit is *node 2*. *Node 2* is a final node, so the visit ends and *node 2* is added to the `pathNodes`. Then the executed path [0, 1, 3, 4, 5, 8, 7, 1, 3, 9, 5, 8, 7, 1, 3, 4, 5, 6, 7, 1, 3, 9, 5, 6, 7, 1, 2] is created.

Chapter 6

Evaluation

In this chapter will be presented the first evaluation test made to *PESTT*. We shown the objectives and expectations, how the assessment process took place and the conclusions drawn. Finally, we present some users testimonials, about their experience of using *PESTT*.

6.1 Evaluation Plan

The objective of this evaluation test is to determine two things. The first is to check the usability of *PESTT*, and the second to check the quality of the produced test.

We conducted the tests on a group of fifteen students attending a graduate course on tests. We divided groups in two: group *A* with eight students and a control group (group *B*) with seven students. All students were familiar with *PESTT* and possess the knowledge required to achieve the evaluation test goals.

For the test we elaborated two simple exercises: exercise *A* consisted in the `remove` method, which removes an element from a list sorted in ascending order, while exercise *B* consisted in the `add` method, which adds an element to the same list, as shown below:

```
1  public class BoundedSortedList<E extends Comparable<E>> {
2    private class Node {...}           // a node in the list
3    // creates a list given its maximum capacity
4    public BoundedSortedList (int capacity) {...}
5    public E get (int i) {...}         // gets the i-th element
6    public int size () {...}     // the size of the list
7    public int capacity () {...}    // the max size of the list
8    public void remove (int i) {  // removes the i-th element
9      Node previous = null;
10     Node current = head;
11     while (current != null && i > 0) {
12       previous = current;
13       current = current.next;
14       i--;
15     }
```

```
16      if (previous == null && current == null)
17        return false;
18      if (i == 0) {
19        if (previous == null && current != null)
20          head = head.next;
21        else if (previous != null && current == null)
22          previous.next = null;
23        else
24          previous.next = current.next;
25      }
26      size--;
27      return true;
28    }
29   // adds a non-existing element v to the list, in ascending
       order
30    public boolean add (E v) {...}
31  }
```

Listing 6.1: Exercise *A*.

```
1   public class BoundedSortedList<E extends Comparable<E>> {
2     private class Node {...}           // a node in the list
3     // creates a list given its maximum capacity
4     public BoundedSortedList (int capacity) {...}
5     public E get (int i) {...}         // gets the i-th element
6     public void remove (int i) {...}   // removes the i-th element
7     public boolean isEmpty () {...}    // is the list empty?
8     public boolean isFull() {...}      // is the list full?
9     // adds a non-existing element v to the list, in ascending
        order
10    public boolean add (E v) {
11      if (isFull())
12        return false;
13
14      Node newNode = new Node ();
15      newNode.v = v;
16
17      Node previous = null;
18      Node current = head;
19
20      while (current != null && current.v.compareTo(v) <= 0) {
21        if (current.v.compareTo(v) == 0)
22          return false;
23        previous = current;
24        current = current.next;
25      }
26
27      newNode.next = current;
28      if (previous == null)
29        head = newNode;
```

```
30      else
31        previous.next = newNode;
32
33      size++;
34      return true;
35    }
36  }
```

Listing 6.2: Exercise *B*.

Exercise *A* was assigned to the group *A* and exercise *B* was assigned to the group *B*. Both groups had a maximum period of 1h30m to perform the following tasks:

1. draw the CFG;
2. obtain the set of requirements based on the Prime Path Coverage criterion;
3. develop the necessary tests, in order to obtain the highest possible level of coverage. This includes two subtasks:

   3.1 identifying the test path required task that had to be performed manually, since the tested version had not yet available the feature of generating test paths automatically; and
   3.2 write the corresponding tests scripts in *Java*;

Group *A* used *PESTT*, while group *B* performed the tasks manually.

Results

In group *A* four students (50%) identified the test paths (the remaining students (the other 50%) not identified any test path). The four students who identified the test paths, three of them (75%) write test for some of the test paths identified. On the other hand, in group *B*, none of the students achieved the last task (were all in the second task).

Then the entire process was repeated switching the groups, i.e. the exercise *A* was assigned to group *B* and the exercise *B* was assigned to group *A*. The tasks and the time were the same, the only difference was this time the group *B* were using the *PESTT* while group *A* performs the tasks manually. This was done to screening the results.

Results

In group *A*, none of the students achieved the last task (were all in the second task). On the other hand, in group *B* all the students (100%) identified the test paths, five of them (71,4%) write test for some of the test paths identified (the remaining students three students (28.6%) not written any test path).

6.2 Conclusions of the Test Evaluation

Initially, our expectations where that at least twelve students (75%) that used *PESTT* (in both groups) would be able to complete all tasks on schedule, for the following reasons:

- they were already familiar with *PESTT*;
- they possess the knowledge required to achieve the evaluation test goals;
- they have sufficient time to achieve the evaluation test goals;
    - the first two tasks would take no more than 10 seconds (since it resumes to click in half a dozen of buttons to performed these tasks);

However, as time passed, our expectations were being undone. This was due to the fact that the students had not much practice identifying the test paths and by presenting unexpected difficulties. Identifying the required test path to get the highest possible level of coverage is a task that had to be performed manually, which took them quite a while, combined with the difficulties presented, only compounded the time taken). Despite this setback, it was possible to identify that the students use *PESTT* without major difficulties, and it was appellant seeing students confirm the values obtained manually in *PESTT*.

The tests wrote by the students correspond to the test paths identified by them. This indicates that at least students understand the relationship between the test paths and the tests that need to be written, focusing only on what really matters.

Although the results have not corroborated our initial expectation, we consider that the overall outcome is very positive, since it is possible to identify students difficulties. This allow us to introduce new features in *PEST*, notably, the generation of test paths automatically. With this new feature available the user's work boils down to the writing of test, all necessary information is provided by *PESTT* automatically and instantly, so that the test engineer can focus only on what really matters: write quality tests instead of not wasting time and resources doing tasks that can be made automatically.

Other of the observed conclusions were the improvement in the students results regarding the processing of mapping a method source code into a CFG manually. It was possible to note that students who had access to *PESTT*, when asked to map a method source code into a CFG manually, showed better results compared with students who had never been in contact with the *PESTT*.

6.3 Users testimonials

We present some comments made by students about *PESTT*.

"(...) PESTT appreciation was very positive, it facilitated the perception of various concepts inherent in the discipline of *Verificação e Validação de Software (SVV)*. More specif-

ically, through its basic features (drawing the graph corresponding to the code method), (...) it is easy to see the major differences between the test concepts, test requirements set, test set paths, inputs, etc... (...) the tool has become a key element in testing activity, (...) it dictated the projects development life-cycle, which could not be changed. (...) determine which criterion we wanted, and generate their test requirements, create the test paths set that would fully cover these requirements, these paths transformed into executable code and finally did the debug to verify the correctness thereof.(...)"

"(...) PESTT is useful to confirm the manually calculations before move forward to later stages where errors have a higher cost. There are small aspects of the graphic interface that I think that could be improved.(...)"

Chapter 7

Conclusion

This thesis addresses the development of the *PESTT Educational Software Testing Tool*. The main objective of this tool is to support teaching, in particular, the introduction of basic concepts and of the different techniques of Software Testing.

The contributions of this work were comprise:

(i) the creation of a plug-in for the *Eclipse* IDE.

(ii) the development of a flexible architecture based on the following modules:

- the Control Flow Graph (CFG) model—the ground of whole project. This module contains the representation of *Java* source code as a CFG model.
- the CFG module—graphical representation of the CFG model using the *Zest* visualization tool.
- the Requirements module—manages the coverage criteria based on CFGs.
- the Test module—manages the test planning and the tests execution.
- the UI module—contains the UI integration with the *Eclipse* IDE, such as, perspectives, views , buttons, etc.

(iii) integration with *JUnit* testing tool to run and obtain the results of tests execution inside the plug-in.

(iv) integration with *Byteman* instrumentation tool to obtain the executed paths, resulting for tests execution.

(v) visual coverage status information.

PESTT was used in a real teaching environment and had a great acceptance (by teachers and students). One of the main points highlighted by teachers was the impact that the use of *PESTT* had on student learning, improving significantly the final results obtained by students. By students the main points highlighted was the easy way of confirming the results—the GFG, the test requirements generated by various criteria, etc.

The implementation of the complete *PESTT* tool, requires engineering work that was deemed incompatible with the duration of this thesis's project, and is subject to follow-up work. The next step may include consolidating and extending the existing features, by including support for report generation and provide more coverage information to the users.

A new evaluation test should be planned and carried out with students, not only to evaluate the new features, but also to be able to compare the quality of the results produced. Another evaluation test should be performed with professionals from the testing area, in order to obtained results and feedback in a real context of use (outside the teaching environment).

Further developments already planned are the implementation of logic coverage criteria and mutation test coverage will make *PESTT* a tool much more powerful and useful. These works are due next June. Write tests to test *PESTT*, writing user manual guide and tutorials are other developments that we intend to do.

More information about *PESTT* can be found in the project's website (`http://pestt.github.com/`).

Appendix A

Pseudocode snippets

A.1 List of rules

```
RULE r1
CLASS DemoClass
METHOD demo
AT ENTRY
IF true
DO System.out.println("0");
ENDRULE

RULE r2
CLASS DemoClass
METHOD demo
AT LINE 4
IF true
DO System.out.println("1 7")
ENDRULE

RULE r3
CLASS DemoClass
METHOD demo
AT LINE 5
IF true
DO System.out.println("3")
ENDRULE

RULE r4
CLASS DemoClass
METHOD demo
AT LINE 6
IF true
DO System.out.println("4")
ENDRULE

RULE r5
```

```
CLASS DemoClass
METHOD demo
AT LINE 8
IF true
DO System.out.println("9")
ENDRULE

RULE r6
CLASS DemoClass
METHOD demo
AT LINE 9
IF true
DO System.out.println("5")
ENDRULE

RULE r7
CLASS DemoClass
METHOD demo
AT LINE 10
IF true
DO System.out.println("6")
ENDRULE

RULE r8
CLASS DemoClass
METHOD demo
AT LINE 12
IF true
DO System.out.println("8")
ENDRULE

RULE r9
CLASS DemoClass
METHOD demo
AT EXIT
IF true
DO System.out.println("2");
ENDRULE
```

Listing A.1: Rules to get node coverage through *Byteman*.

The list of rules used in the example illustrated in 3.2.3.

A.2 MANIFEST.MF

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: PESTT
Bundle-SymbolicName: PESTT;singleton:=true
```

```
Bundle-Version: 0.4
Bundle-Activator: main.activator.Activator
Bundle-RequiredExecutionEnvironment: JavaSE-1.6,
 JavaSE-1.7
Import-Package: org.eclipse.ui.texteditor
Bundle-ActivationPolicy: lazy
Export-Package: adt.graph;uses:="domain.graph.visitors",
 domain;
  uses:="domain.graph.visitors,
   org.eclipse.jdt.core.dom,
   domain.controllers,
   org.eclipse.zest.core.widgets,
   adt.graph,
   org.eclipse.jdt.core,
   domain.coverage.algorithms,
   ui.controllers,
   domain.coverage.data",
 domain.constants,
 domain.controllers;
  uses:="domain.graph.visitors,
   org.eclipse.jdt.core.dom,
   adt.graph,
   org.eclipse.jdt.core,
   domain.constants,
   domain,
   domain.coverage.data",
 domain.coverage.algorithms;uses:="adt.graph,domain.constants,
    domain",
 domain.coverage.data;uses:="adt.graph",
 domain.coverage.instrument;uses:="org.eclipse.jdt.junit.
    launcher,org.eclipse.ui,org.eclipse.jdt.junit",
 domain.dot.processor,
 domain.events;uses:="adt.graph,domain.constants",
 domain.explorer;uses:="org.eclipse.jdt.core.dom,adt.graph",
 domain.graph.visitors;uses:="adt.graph",
 main.activator;
  uses:="domain.controllers,
   org.eclipse.jface.resource,
   org.eclipse.ui.plugin,
   org.osgi.framework,
   ui.controllers",
 org.jboss.byteman.agent,
 org.jboss.byteman.agent.adapter,
 org.jboss.byteman.agent.adapter.cfg,
 org.jboss.byteman.agent.check,
 org.jboss.byteman.java_cup.runtime,
 org.jboss.byteman.objectweb.asm,
 org.jboss.byteman.objectweb.asm.commons,
 org.jboss.byteman.objectweb.asm.signature,
```

```
org.jboss.byteman.objectweb.asm.tree,
org.jboss.byteman.objectweb.asm.tree.analysis,
org.jboss.byteman.objectweb.asm.util,
org.jboss.byteman.objectweb.asm.xml,
org.jboss.byteman.rule,
org.jboss.byteman.rule.binding,
org.jboss.byteman.rule.compiler,
org.jboss.byteman.rule.exception,
org.jboss.byteman.rule.expression,
org.jboss.byteman.rule.grammar,
org.jboss.byteman.rule.helper,
org.jboss.byteman.rule.type,
org.jboss.byteman.synchronization,
org.jboss.byteman.test,
ui,
ui.constants;uses:="org.eclipse.swt.graphics",
ui.controllers;
 uses:="ui.editor,
  ui,
  org.eclipse.jdt.core,
  domain.constants,
  org.eclipse.ui,
  ui.source",
ui.dialog;uses:="org.eclipse.swt.widgets",
ui.display.perspective;uses:="org.eclipse.ui",
ui.display.views;uses:="org.eclipse.ui.part,org.eclipse.swt.
   widgets",
ui.display.views.logic;uses:="org.eclipse.swt.widgets",
ui.display.views.structural;
 uses:="ui.constants,
  org.eclipse.ui,
  org.eclipse.jface.viewers,
  org.eclipse.swt.widgets",
ui.display.views.structural.defuses;
 uses:="org.eclipse.ui,
  org.eclipse.jface.viewers,
  org.eclipse.swt.widgets,
  ui.display.views.structural",
ui.editor;uses:="org.eclipse.jdt.core,org.eclipse.ui,org.
   eclipse.core.resources",
ui.events;uses:="domain.constants",
ui.handler;uses:="org.eclipse.core.commands",
ui.preferences;uses:="org.eclipse.jface.preference,org.eclipse.
   ui,org.eclipse.core.runtime.preferences",
ui.source;
 uses:="org.eclipse.zest.layouts,
  domain.constants,
  org.eclipse.swt.graphics,
  org.eclipse.zest.layouts.interfaces,
```

```
98     org.eclipse.swt.widgets,
99     domain.coverage.data"
100 Bundle-ClassPath: .,
101  lib/byteman.jar
102 Require-Bundle: org.eclipse.ui;bundle-version="3.7.0",
103  org.eclipse.zest.dot.core;bundle-version="2.0.0",
104  org.eclipse.ui.console;bundle-version="3.5.100",
105  org.eclipse.jdt.launching;bundle-version="3.6.0",
106  org.eclipse.debug.ui,
107  org.eclipse.jdt.junit;bundle-version="3.7.0"
```

Listing A.2: `MANIFEST.MF` file content.

Line 1 to 5 describe *PESTT* general characteristics. Line 6 indicates the plug-in activator. Lines 7–8 specifies the supported *Java* version. Line 9 contains the imported package. Line 10 is the activator policy used. Line 11 to 99 indicates the exported packages and their uses. Lines 100–101 define the plug-in classpath. Finally, lines 102 to 107 enumerate the plug-in dependencies.

A.3 `plugin.xml`

```
1  <extension
2    point="org.eclipse.ui.preferencePages">
3    <page
4      class="ui.preferences.PreferencePage"
5      id="preferences.PreferencePage"
6      name="PESTT">
7    </page>
8  </extension>
9  <extension
10   point="org.eclipse.core.runtime.preferences">
11   <initializer
12     class="ui.preferences.PreferenceInitializer">
13   </initializer>
14 </extension>
15 <extension
16   point="org.eclipse.ui.perspectives">
17   <perspective
18     class="ui.display.perspective.Perspective"
19     fixed="false"
20     icon="icons/PESTT.gif"
21     id="PESTT.Perspective"
22     name="PESTT">
23   </perspective>
24 </extension>
25 <extension
26   point="org.eclipse.ui.views">
27   <category
```

```
28      id="PESTT.Category"
29      name="PESTT">
30    </category>
31    <view
32      allowMultiple="true"
33      category="PESTT.Category"
34      Class="ui.display.views.ViewGraph"
35      icon="icons/graph.gif"
36      id="PESTT.ViewGraph"
37      name="Graph"
38      restorable="true">
39    </view>
40  </extension>
41  <extension
42    point="org.eclipse.ui.perspectiveExtensions">
43    <perspectiveExtension
44      targetID="PESTT.Perspective">
45      <view
46        closeable="true"
47        id="PESTT.ViewGraph"
48        minimized="false"
49        moveable="true"
50        relationship="right"
51        relative="org.eclipse.ui.editors"
52        showTitle="true"
53        standalone="false"
54        visible="true">
55      </view>
56    </perspectiveExtension>
57  </extension>
58  <extension
59    point="org.eclipse.ui.commands">
60    <category
61     description="Commands related to the PESTT Perspective."
62     id="PESTT.Category"
63      name="PESTT">
64    </category>
65    <command
66      categoryId="PESTT.Category"
67      description="Draw the graph."
68      id="PESTT.DrawGraph"
69      name="DrawGraph">
70    </command>
71  </extension>
```

Listing A.3: Excerpt from the content of the `plugin.xml` file.

Lines 1 to 8 define the plug-in Preference Page. Lines 9 to 14 describe plug-in preferences. Line 15 to 24 include the plug-in perspective definition. Lines 25 to 40 is the

Graph view definition. Lines 41 to 57 represent the Graph view position inside the plug-in perspective. Finally, lines 59 to 71 define the draw graph button.

A.4 Graph Coverage Criteria View

```
1  @SuppressWarnings("deprecation")
2  public class GraphCoverageCriteria implements Observer {
3
4    private Graph graph;
5    private Map<GraphCoverageCriteriaId, GraphNode> nodes;
6    private SelectionAdapter event;
7
8    public GraphCoverageCriteria(Composite parent) {
9      graph = new Graph(parent, SWT.NONE);
10     Activator.getDefault().getTestRequirementController().
         addObserver(this);
11     Activator.getDefault().getCFGController().addObserver(this);
12     create();
13   }
14
15   private void create() {
16     graph.clear();
17     removeSelectionListener();
18     setNodes();
19     setEdges();
20     setLayout();
21     addSelectionListener();
22     if(Activator.getDefault().getTestRequirementController().
         isCoverageCriteriaSelected())
23       setSelected(nodes.get(Activator.getDefault().
           getTestRequirementController().
           getSelectedCoverageCriteria()));
24   }
25
26   public void dispose() {
27     Activator.getDefault().getTestRequirementController().
         deleteObserver(this);
28     Activator.getDefault().getCFGController().deleteObserver(
         this);
29   }
30
31   private void setNodes() {
32     nodes = new HashMap<GraphCoverageCriteriaId, GraphNode>();
33     GraphNode cpc = new GraphNode(graph, SWT.SINGLE, "Complete
         Path\n      Coverage\n" + insertTrace(14) + "\n
                   (CPC)");
34     cpc.setData(GraphCoverageCriteriaId.COMPLETE_PATH);
```

```
35      cpc.setTooltip(new Label("Complete Path Coverage (CPC):\
           nTest requirements contains all paths in Graph."));
36      nodes.put(GraphCoverageCriteriaId.COMPLETE_PATH, cpc);
37
38      GraphNode ppc = new GraphNode(graph, SWT.SINGLE, "Prime Path
           \n Coverage\n" + insertTrace(11) + "\n      (PPC)");
39      ppc.setData(GraphCoverageCriteriaId.PRIME_PATH);
40      ppc.setTooltip(new Label("Prime Path Coverage (PPC):\nTest
           requirements contains each prime path in Graph."));
41      nodes.put(GraphCoverageCriteriaId.PRIME_PATH, ppc);
42
43      GraphNode adupc = new GraphNode(graph, SWT.SINGLE, "All-du-
           Paths\n  Coverage\n " + insertTrace(11) + "\n    (ADUPC)"
           );
44      adupc.setData(GraphCoverageCriteriaId.ALL_DU_PATHS);
45      adupc.setTooltip(new Label("All-du-Paths Coverage (ADUPC):\
           nFor each def-pair set S = du(ni, nj, v),\nTest
           requirements contains every path d in S."));
46      nodes.put(GraphCoverageCriteriaId.ALL_DU_PATHS, adupc);
47
48      GraphNode epc = new GraphNode(graph, SWT.SINGLE, "Edge-Pair\
           nCoverage\n" + insertTrace(9) + "\n     (EPC)");
49      epc.setData(GraphCoverageCriteriaId.EDGE_PAIR);
50      epc.setTooltip(new Label("Edge-Pair Coverage (EPC):\nTest
           requirements contains each reachable path of length up to
            2, inclusive, in Graph."));
51      nodes.put(GraphCoverageCriteriaId.EDGE_PAIR, epc);
52
53      GraphNode crtc = new GraphNode(graph, SWT.SINGLE, "Complete
           Round\n   Trip Coverage\n" + insertTrace(16) + "\n
                     (CRTC)");
54      crtc.setData(GraphCoverageCriteriaId.COMPLETE_ROUND_TRIP);
55      crtc.setTooltip(new Label("Complete Round Trip Coverage (
           CRTC):\nTest requirements contains all round-trip paths
           for each reachable node in Graph."));
56      nodes.put(GraphCoverageCriteriaId.COMPLETE_ROUND_TRIP, crtc)
           ;
57
58      GraphNode auc = new GraphNode(graph, SWT.SINGLE, "  All-Uses
           \nCoverage\n" + insertTrace(9) + "\n    (AUC)");
59      auc.setData(GraphCoverageCriteriaId.ALL_USES);
60      auc.setTooltip(new Label("All-Uses Coverage (AUC):\nFor each
            def-pair set S = du(ni, nj, v),\nTest requirements
           contains at least one path d in S."));
61      nodes.put(GraphCoverageCriteriaId.ALL_USES, auc);
62
63      GraphNode ec = new GraphNode(graph, SWT.SINGLE, "     Edge\n
            Coverage\n" + insertTrace(10) + "\n      (EC)");
64      ec.setData(GraphCoverageCriteriaId.EDGE);
```

```
65      ec.setTooltip(new Label("Edge Coverage (EC):\nTest
           requirements contains each reachable path of length up to
            1, inclusive, in Graph."));
66      nodes.put(GraphCoverageCriteriaId.EDGE, ec);
67
68      GraphNode srtc = new GraphNode(graph, SWT.SINGLE, "Simple
           Round\nTrip Coverage\n" + insertTrace(13) + "\n        (
           SRTC)");
69      srtc.setData(GraphCoverageCriteriaId.SIMPLE_ROUND_TRIP);
70      srtc.setTooltip(new Label("Simple Round Trip Coverage (SRTC)
           :\nTest requirements contains at least one round-trip
           path\n for each reachable node in Graph that begins and
           ends a round-trip path."));
71      nodes.put(GraphCoverageCriteriaId.SIMPLE_ROUND_TRIP, srtc);
72
73      GraphNode adc = new GraphNode(graph, SWT.SINGLE, "  All-Defs
           \nCoverage\n" + insertTrace(9) + "\n    (ADC)");
74      adc.setData(GraphCoverageCriteriaId.ALL_DEFS);
75      adc.setTooltip(new Label("All-Defs Coverage (ADC):\nFor each
            def-path set S = du(n, v),\nTest requirements contains
           at least one path d in S."));
76      nodes.put(GraphCoverageCriteriaId.ALL_DEFS, adc);
77
78      GraphNode nc = new GraphNode(graph, SWT.SINGLE, "    Node\
           nCoverage\n" + insertTrace(9) + "\n     (NC)");
79      nc.setData(GraphCoverageCriteriaId.NODE);
80      nc.setTooltip(new Label("Node Coverage (NC):\nTest
           requirements contains each reachable node in Graph."));
81      nodes.put(GraphCoverageCriteriaId.NODE, nc);
82
83      for(GraphNode gnode : nodes.values()) {
84        gnode.setBackgroundColor(Colors.WHITE);
85        gnode.setForegroundColor(Colors.BLACK);
86        gnode.setBorderColor(Colors.BLACK);
87        gnode.setHighlightColor(Colors.YELLOW);
88        gnode.setBorderHighlightColor(Colors.BLACK);
89      }
90    }
91
92    private void setEdges() {
93      new GraphConnection(graph, ZestStyles.CONNECTIONS_DIRECTED,
           nodes.get(GraphCoverageCriteriaId.COMPLETE_PATH), nodes.
           get(GraphCoverageCriteriaId.PRIME_PATH));
94      new GraphConnection(graph, ZestStyles.CONNECTIONS_DIRECTED,
           nodes.get(GraphCoverageCriteriaId.PRIME_PATH), nodes.get(
           GraphCoverageCriteriaId.ALL_DU_PATHS));
95      new GraphConnection(graph, ZestStyles.CONNECTIONS_DIRECTED,
           nodes.get(GraphCoverageCriteriaId.PRIME_PATH), nodes.get(
           GraphCoverageCriteriaId.EDGE_PAIR));
```

```
96      new GraphConnection(graph, ZestStyles.CONNECTIONS_DIRECTED,
            nodes.get(GraphCoverageCriteriaId.PRIME_PATH), nodes.get(
            GraphCoverageCriteriaId.COMPLETE_ROUND_TRIP));
97      new GraphConnection(graph, ZestStyles.CONNECTIONS_DIRECTED,
            nodes.get(GraphCoverageCriteriaId.ALL_DU_PATHS), nodes.
            get(GraphCoverageCriteriaId.ALL_USES));
98      new GraphConnection(graph, ZestStyles.CONNECTIONS_DIRECTED,
            nodes.get(GraphCoverageCriteriaId.EDGE_PAIR), nodes.get(
            GraphCoverageCriteriaId.EDGE));
99      new GraphConnection(graph, ZestStyles.CONNECTIONS_DIRECTED,
            nodes.get(GraphCoverageCriteriaId.COMPLETE_ROUND_TRIP),
            nodes.get(GraphCoverageCriteriaId.SIMPLE_ROUND_TRIP));
100     new GraphConnection(graph, ZestStyles.CONNECTIONS_DIRECTED,
            nodes.get(GraphCoverageCriteriaId.ALL_USES), nodes.get(
            GraphCoverageCriteriaId.EDGE));
101     new GraphConnection(graph, ZestStyles.CONNECTIONS_DIRECTED,
            nodes.get(GraphCoverageCriteriaId.ALL_USES), nodes.get(
            GraphCoverageCriteriaId.ALL_DEFS));
102     new GraphConnection(graph, ZestStyles.CONNECTIONS_DIRECTED,
            nodes.get(GraphCoverageCriteriaId.EDGE), nodes.get(
            GraphCoverageCriteriaId.NODE));
103   }
104
105   private String insertTrace(int num) {
106     String line = "";
107     for(int i = 0; i < num; i++)
108       line += (char)0x2013 + "";
109     return line;
110   }
111
112   private void setLayout() {
113     graph.setLayoutAlgorithm(new TreeLayoutAlgorithm(
            LayoutStyles.NO_LAYOUT_NODE_RESIZING), true);
114   }
115
116   private void addSelectionListener() {
117     event = new SelectionAdapter() { // create a new
            SelectionAdapter event.
118
119       @Override
120       public void widgetSelected(SelectionEvent e) {
121         if(e.item != null && e.item instanceof GraphNode ) {
122           setSelected(null);
123           setSelected((GraphItem) e.item);
124           GraphCoverageCriteriaId option = (
                GraphCoverageCriteriaId) getSelected().getData();
125           Activator.getDefault().getTestRequirementController().
                selectCoverageCriteria(option);
126         } else
```

```
127             Activator.getDefault().getTestRequirementController().
                    selectCoverageCriteria(null);
128         }
129       };
130       graph.addSelectionListener(event);
131     }
132
133     private void removeSelectionListener() {
134       if(event != null)
135         graph.removeSelectionListener(event);
136     }
137
138     private GraphItem getSelected() {
139       if(graph.getSelection().size() == 0)
140         return null;
141       return (GraphItem) graph.getSelection().get(0); // return
              the list with the selected nodes.
142     }
143
144     private void setSelected(GraphItem item) {
145       graph.setSelection(item == null ? null : new GraphItem[] {
              item}); // the items selected.
146     }
147
148     @Override
149     public void update(Observable obs, Object data) {
150       if(data instanceof TestRequirementSelectedCriteriaEvent)
151         setSelected(nodes.get(Activator.getDefault().
                getTestRequirementController().
                getSelectedCoverageCriteria()));
152       else if(data instanceof RefreshStructuralGraphEvent)
153         create();
154     }
155 }
```

Listing A.4: Graph Coverage Criteria file.

This class creates the Graph that is shown in the Structural Coverage view using the *Zest* visualization tool (Line 9). Method `setNodes` creates the graph nodes corresponding to each coverage criteria. Method `setEdges` sets the connections (graph edges) between nodes. The `insertTrace` method inserts a trace in graph nodes—to separate the full coverage criteria name from their acronym. The `setLayout` method sets the graph layout used. Methods `addSelectionListener` and `removeSelectionListener` add and remove graph, a listeners respectively to be notified when the user selects a graph element. The `getSelected` method returns the selected graph element. Method `setSelected` selects a graph element. The `update` dispatches external notifications.

A.5 CFG Builder

```
1  ForStatement [114, 265]
2    INITIALIZERS (1)
3    EXPRESSION
4    UPDATERS (1)
5    BODY
6      Block [142, 237]
7        STATEMENTS (2)
8          IfStatement [150, 96]
9            EXPRESSION
10             InfixExpression [153, 10]
11               > (Expression) type binding: boolean
12               Boxing: false; Unboxing: false
13               ConstantExpressionValue: null
14               LEFT_OPERAND
15               OPERATOR: '=='
16               RIGHT_OPERAND
17               EXTENDED_OPERANDS (0)
18           THEN_STATEMENT
19             ExpressionStatement [173, 27]
20               EXPRESSION
21                 MethodInvocation [173, 26]
22                   > (Expression) type binding: void
23                   > method binding: PrintStream.println(String)
24                   ResolvedTypeInferredFromExpectedType: false
25                   Boxing: false; Unboxing: false
26                   ConstantExpressionValue: null
27                   EXPRESSION
28                   TYPE_ARGUMENTS (0)
29                   NAME
30                   ARGUMENTS (1)
31           ELSE_STATEMENT
32             ExpressionStatement [220, 26]
33               EXPRESSION
34         IfStatement [253, 122]
35           EXPRESSION
36           THEN_STATEMENT
37           ELSE_STATEMENT
```

Listing A.5: AST excerpt for the method in Listing 3.1

This listing describes the *AST* internal organization for the method in Listing 3.1. The *AST* represents the *Java* source code information hierarchically structured. The CFG for the method is build as the *AST* is traversed using a depth first algorithm. Lines 6 to 37 represent the **for** statement body, which contains two **if** statements. On its turn, the first **if** statement contains three sub nodes: the guard expression, the **if**-then statement, and the **else** statement.

A.6 Users testimonials

Comments made by students about *PESTT*.

"A minha apreciação da ferramenta *PESTT* foi bastante positiva, dado que facilitou a perceção de diversos conceitos inerentes à cadeira de *Verificação e Validação de Software (VVS)*. Mais concretamente, através da sua funcionalidade base (desenho do grafo correspondente ao código em questão), foi-nos possível perceber melhor como era feita a relação entre um troço de código e o seu grafo correspondente, fazendo com que os próprios alunos cometessem menos erros ao desenhar os grafos por si mesmos. Adicionalmente, tornou-se fácil perceber as principais diferenças entre os conceitos de teste, requisitos de teste, conjunto de caminhos de teste, inputs, etc... Isto porque recorrendo ao *PESTT*, esses mesmos conceitos eram postos em prática, pelo que fazia com que os próprios alunos estudassem (de certo modo) a matéria referente a *VVS*, aquando do desenvolvimento dos projetos. Finalmente e do meu ponto de vista, a ferramenta tornou-se um elemento chave na atividade de teste, porque determinou (falo por experiência própria) e ditou para cada um dos projectos, um ciclo de desenvolvimento, o qual não podia ser alterado. Mais especificamente, considero que de certo modo a ferramenta fez com que eu soubesse que teria de determinar qual o critério que queria, gerasse os seus requisitos de teste, criasse um conjunto de caminhos de teste que cobrissem na totalidade esses requisitos, transformasse esses caminhos em código executável e finalmente fizesse o debug para verificar a correcção dos mesmos. Tornou-se portanto extremamente útil e uma ferramenta sem a qual não seria tão fácil de desenvolver os referidos projectos."

"Achei o *PESTT* muito interessante. Utilizei-o como ferramenta de estudo. Mesmo quando tem que se saber fazer os algoritmos em papel, o *PESTT* é útil para confirmar os cálculos manuais, ajudando a que possamos confirmar se os resultados obtidos até certo ponto de uma resolução estão correctos, antes de avançar para fases posteriores em que as incorrecções teriam um custo mais alto. Existem pequenos aspectos do interface gráfico que penso que podiam ser melhorados. Nomeadamente, a inserção de um caminho de teste através de uma caixa de texto não é muito intuitiva. Seria mais fácil fazê-lo directamente sobre o grafo. Seria também interessante que o plug-in conseguisse acompanhar a execução de um script de testes, como acontece nas ferramentas de coverage, e identificasse os caminhos de forma automática."

A.7 Users testimonials

Id	*Test requirement*	*Id*	*Test requirement*
1	[0, 1, 2]	*22*	[5, 6, 7, 1, 3, 4, 5]
2	[0, 1, 3, 4, 5, 6, 7]	*23*	[5, 6, 7, 1, 3, 9, 5]
3	[0, 1, 3, 4, 5, 8, 7]	*24*	[5, 8, 7, 1, 3, 4, 5]
4	[0, 1, 3, 9, 5, 6, 7]	*25*	[5, 8, 7, 1, 3, 9, 5]
5	[0, 1, 3, 9, 5, 8, 7]	*26*	[6, 7, 1, 3, 4, 5, 6]
6	[1, 3, 4, 5, 6, 7, 1]	*27*	[6, 7, 1, 3, 4, 5, 8]
7	[1, 3, 4, 5, 8, 7, 1]	*28*	[6, 7, 1, 3, 9, 5, 6]
8	[1, 3, 9, 5, 6, 7, 1]	*29*	[6, 7, 1, 3, 9, 5, 8]
9	[1, 3, 9, 5, 8, 7, 1]	*30*	[7, 1, 3, 4, 5, 6, 7]
10	[3, 4, 5, 6, 7, 1, 2]	*31*	[7, 1, 3, 4, 5, 8, 7]
11	[3, 4, 5, 6, 7, 1, 3]	*32*	[7, 1, 3, 9, 5, 6, 7]
12	[3, 4, 5, 8, 7, 1, 2]	*33*	[7, 1, 3, 9, 5, 8, 7]
13	[3, 4, 5, 8, 7, 1, 3]	*34*	[8, 7, 1, 3, 4, 5, 6]
14	[3, 9, 5, 6, 7, 1, 2]	*35*	[8, 7, 1, 3, 4, 5, 8]
15	[3, 9, 5, 6, 7, 1, 3]	*36*	[8, 7, 1, 3, 9, 5, 6]
16	[3, 9, 5, 8, 7, 1, 2]	*37*	[8, 7, 1, 3, 9, 5, 8]
17	[3, 9, 5, 8, 7, 1, 3]	*38*	[9, 5, 6, 7, 1, 3, 4]
18	[4, 5, 6, 7, 1, 3, 4]	*39*	[9, 5, 6, 7, 1, 3, 9]
19	[4, 5, 6, 7, 1, 3, 9]	*40*	[9, 5, 8, 7, 1, 3, 4]
20	[4, 5, 8, 7, 1, 3, 4]	*41*	[9, 5, 8, 7, 1, 3, 9]
21	[4, 5, 8, 7, 1, 3, 9]		

Table A.1: The set of requirement paths generated to the prime path coverage criterion for Listing 3.1

Bibliography

[1] Google CodePro Analytix. Coverage analysis tool. https://developers.google.com/java-dev-tools/codepro/doc/.

[2] Jeff McAffer & Paul VanderLei & Simon Archer. *"OSGi And Equinox: Creating Highly Modular Java Systems"*. Addison-Wesley Professional, February 2010.

[3] ASM. Java bytecode manipulation and analysis framework. http://asm.ow2.org/.

[4] Kent Beck. Kent Beck's page at the Three Rivers Institute. http://www.threeriversinstitute.org/Kent%20Beck.htm.

[5] Ilene Burnstein. *"Practical Software Testing"*. Springer, 2003.

[6] Byteman. A tool which simplifies tracing and testing of Java programs. http://www.jboss.org/byteman/.

[7] Clover. Coverage analysis tool. http://www.atlassian.com/software/clover/overview.

[8] Cobertura. Coverage analysis tool. http://cobertura.sourceforge.net/index.html.

[9] CodeCover. Coverage analysis tool. http://codecover.org/index.html.

[10] JMockit Coverage. Coverage analysis tool. http://jmockit.googlecode.com/svn/trunk/www/tutorial/CodeCoverage.html.

[11] DevPartner. Coverage analysis tool. http://www.microfocus.com/products/micro-focus-developer/devpartner/index.aspx.

[12] EclEmma. Coverage analysis tool. http://www.eclemma.org/.

[13] Emma. Coverage analysis tool. http://emma.sourceforge.net/.

[14] *Elipse Project*. Development Works http://www.eclipse.org/.

[15] Equinox. *Eclipse* certified implementation of the OSGi R4.x core framework specification. http://www.eclipse.org/equinox/.

[16] Erich Gamma. Erich Gamma wikipedia information. http://en.wikipedia.org/wiki/Erich_Gamma.

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *"Design Patterns: Elements of Reusable Object-Oriented Software"*. Addison-Wesley Professional, November 1994.

[18] GEF. Graphical Editing Framework for *Eclipse*. http://www.eclipse.org/gef/.

[19] Eclipse Control Flow Graph Generator. *Eclipse* plug-in for generating Control Flow Graphs. http://eclipsefcg.sourceforge.net/.

[20] Graphviz. Graph visualization *software*. http://www.graphviz.org/.

[21] IBM. Development Works http://www.ibm.com/developerworks/opensource/library/os-ecjdt/.

[22] Instrumentation. types of instrumentation. https://confluence.atlassian.com/pages/viewpage.action?pageId=79986998.

[23] JaCoCo. Java code coverage library. http://www.eclemma.org/jacoco/trunk/index.html.

[24] Nick Jenkins. *"A Software Testing Primer"*. Creative Commons, 2008.

[25] Jtest. Unit testing *software* framework for the *Java* programming language. http://www.parasoft.com/jsp/products/jtest.jsp?itemId=14.

[26] JUnit. Unit testing *software* framework for the *Java* programming language. http://www.junit.org/.

[27] Eclipse Public License. Eclipse Public License. http://www.eclipse.org/legal/epl-v10.html.

[28] Model-View-Controller. Model-View-Controller details. http://msdn.microsoft.com/en-us/library/ff649643.aspx.

[29] NUnit. Unit testing *software* framework for the *.Net* programming language. http://www.nunit.org/.

[30] Paul Ammann & Jeff Offutt. *"Introduction to Software Testing"*. Cambridge, 2008. http://cs.gmu.edu/~offutt/softwaretest/.

[31] OSGi. Open Services Gateway initiative framework. http://www.osgi.org/Main/HomePage.

[32] Design patterns. Design patterns details. http://sourcemaking.com/design_patterns.

[33] Eric Clayberg & Dan Rubel. *"eclipse Plug-ins"*. Addison-Wesley Professional, 3rd edition, December 2008.

[34] Smalltalk. Object-oriented, dynamically typed, reflective programming language. http://www.world.st/.

[35] SUnit. Unit testing *software* framework for the *Smalltalk* programming language. http://sunit.sourceforge.net/.

[36] TestNG. Unit testing *software* framework for the *Java* programming language. http://testng.org/doc/index.html.

[37] Zest. Visualization toolkit for *Eclipse*. http://www.eclipse.org/gef/zest/index.php.